Objektové modelování pomocí UML v praxi, 2005

díl 1

PDF e-kniha

© RNDr. Ilja Kraval, autor, mailto:objects@objects.cz, leden 2005

Obsah:

1. Základní pojmy z modelování informačních systémů

1.1 Proč vlastně modelovat?

Jako bývalý fyzik jsem jednu dobu pracoval ve vývojovém oddělení závodu na výrobu mikročipů. Ve výrobním závodě vůbec nepřipadá v úvahu, aby pracovník vzal do ruky nějaký rozdělaný výrobek, sám posoudil, co by se tak asi s ním mělo dělat a postupoval dále ve výrobě podle svého uvážení. Ve výrobě jsou na všechno přesně stanovené postupky a nelze proto provést s výrobkem libovolnou operaci podle momentální nálady pracovníka. V „klasickém“ výrobním závodě na všechno včetně detailů existují postupky a návodky, které pracovníci dodržují. Skutečnost, že je třeba v závodě postupovat podle návodek a předpisů, je mimo jakoukoliv diskusi.

Proto jsem zažil poměrně dost velký šok, když jsem po zkušenostech ve výrobním závodě nastoupil do jedné nejmenované soukromé softwarové firmy jako analytik. Tato firma vyráběla bankovní software. Postupky a návodky v ní neexistovaly. Pracovníci volili řešení „ad hoc“ na místě takové, jaké se jim v dané chvíli jevilo jako nejvhodnější. Předávání výsledků tvorby softwaru vůbec neexistovalo, protože každý pracovník tvořil pouze svoji agendu. Pokud se výjimečně předávaly výsledky práce, tak pouze ústní formou. Stejně tak zadání práce, tj. „co se má dělat“, nemělo vůbec žádný formální charakter. Zadání prací bylo neurčité, nepřesné a mnohdy dokonce nebylo ani zaznamenané.

Můj osobní pocit při změně prostředí z výrobního závodu do softwarové firmy byl velmi nepříjemný: V softwarové firmě vládl chaos, tvořil se chybový software, převládala nekvalita, každý tvořil software podle svých postupů...

Někdo by mohl namítnout, že tvorba softwaru a výroba mikročipů jsou natolik odlišné oblasti, že srovnání není na místě. Jedno je přece „kreativní tvorba“ vyžadující velkou tvůrčí svobodu a druhé je „výroba“ vyžadující přísné postupy včetně bezpečnosti práce. Bohužel je třeba oponovat. Dávno pryč jsou ty doby, kdy zdrojové kódy psal nějaký napůl geniální „programátor - vlk samotář“ za bezesných nocí. Tvorba softwaru je výrobou jako každá jiná výroba. Proto podléhá stejným ekonomickým zákonitostem. Dodaný software (například IS na klíč) je normálním předmětem výroby jako každý jiný produkt a proto jeho tvorba podléhá stejným „ekonomickým“ zákonitostem.

Software je věc vysoce abstraktní a „nehmotná“. Z toho důvodu naopak vyžaduje ještě přísnější pravidla než výroba „hmotných věcí“. Pokud například soustružník dostane za úkol vysoustružit kužel a odevzdá válec, tak všichni vidí zmetek hned na první pohled. U softwaru je situace trochu složitější, protože povaha výrobku je mnohem abstraktnější.

Z uvedeného vyplývá, že při tvorbě softwaru musí existovat pravidla, která budou schopna zvládat výrobu pomocí jednotných postupů. Výstupní kontrola softwaru by

měla probíhat velmi podobně, jako když soustružník bere do ruky šupleru a kontroluje rozměry výrobku proti dodanému rysu. Stejně tak pracovník softwarové firmy by měl „proměřit“ zhotovenou část softwaru. Je otázkou, co při tomto „měření“ vlastně může s čím porovnat, když chce posoudit správnost výstupů výroby. Ve většině softwarových firem nelze díky chaotickému vývoji kvalitativně určit, zda to, co se vyrábí, je opravdu tím, co se žádá vyrobit. Paradoxně rozdíl povahy softwaru jako výrobku velmi „nehmotného“ potřebuje mnohem důslednější dodržování pravidel výroby včetně výstupní kontroly. Dokonce to vyžaduje paradoxně mnohem více důslednosti než u výrobků více hmatatelných.

1.2 Metoda výroby softwaru zvaná TUNEL

Aniž bych to v té době věděl, byl jsem ve zmíněné firmě svědkem použití metody tvorby softwaru nazývané v literatuře příznačně jako „metoda TUNEL“. Tato metoda je charakterizována tím, že na počátku projektu se vstupuje do černého tunelu, kterým se prochází poslepu od stěny ke stěně. Operativními zásahy (tj. nárazy do stěn tunelu) se vedoucí projektu snaží projekt uřídit a najít světýlko na konci tunelu. Mnohdy se projekt „s odřenýma ušima a vyplazeným jazykem“ dokončí a „jakýs takýs“ informační systém se zákazníkovi nakonec odevzdá. Bohužel mnohdy se nadějné světélko na konci tunelu změní na světla protijedoucího vlaku a celý projekt skončí katastrofickým scénářem.

Metoda TUNEL má své charakteristické rysy:

- Chybí jakákoliv koncepce vývoje a není možná opakovatelnost výsledků. V metodě TUNEL nejsou použity žádné opakovatelné postupy a projekt se řídí pouze momentální intuicí a zkušenostmi vedoucího projektu. Každý projekt a jeho výstupy vypadají jinak, a to dokonce i v případě, že projekty řídí jeden a tentýž vedoucí. Výsledky z různých projektů jsou různé, jsou nesourodé, navzájem si odporují, apod.
- Není možná jakákoliv predikce v projektu. Ani na začátku, ani v průběhu projektu, a to dokonce ani v jeho závěrečných fázích nelze odhadovat další pracnost, tj. jaké všechny práce bude třeba ještě udělat. Účastníci projektu nevědí, co je čeká na jejich cestě „za příštím rohem“. Každý projekt se tak stává sázkou do loterie a připomíná spíše dobrodružnou výpravu za pokladem pirátů než výrobu softwaru.
- Platí obecná neznalost kdy má co kdo dělat. Vše je pouze v kompetenci vedoucího projektu, který řídí projekt spíše ze zkušeností a pouze pomocí vlastní intuice. Vedoucí nemá žádnou příručku, ani žádné rady a pokyny, jak

v dané chvíli postupovat, co vyžadovat. Proto volí své vlastní ať už lepší nebo horší řešení přímo v dané chvíli a na daném místě. Řízení se pro něj stává velmi náročnou prací plnou lokálních operativních zásahů bez možnosti jakékoliv kontroly správnosti rozhodnutí, přičemž z hlediska metod řízení chybí jakékoliv kvalitativní porovnání čehokoliv s čímkoliv.

- Každá porada v projektu začíná „jobovými zvěstmi“, co vše nechodí, co chybí, co je třeba ještě udělat. Vedoucí projektu, který používá metodu TUNEL, většinou mívá žaludeční vředy.
- Ve výstupech z projektů, které se průběžně tvoří metodou TUNEL, neexistuje požadavek na opětovnou použitelnost neboli re-use, anebo přesněji řečeno, při metodě TUNEL se tento požadavek nedá efektivně zavést. Některé části agend se vyvíjejí několikrát, což vede k dalším nečekaným problémům díky existenci několikanásobných řešení. Nejenže se bez zavedení opětovné použitelnosti jedná při opakování vývoje o zbytečnou práci, ale software začíná vykazovat velmi silnou nepřehlednost. Jednotlivé prvky softwaru v projektech se díky své násobnosti velmi těžko identifikují. Neví se, kde všude se opakující chyba resp. požadavek na změnu vlastně vyskytuje. To se projevuje jako závažný nedostatek zejména při opravách a případných změnách. Opravdu velmi obtížně (někdy dokonce až za hranicí možného) se vyhledávají všechna místa, kde se všude chyba nebo požadavek na změnu nachází. V některých případech se při opravě chyby u hodně složitých a rozsáhlých systémů dokonce ani nenajdou všechna místa, kde se chyba vyskytuje, protože se o všech místech prostě neví. Systém vykazuje velmi nízkou transparenci, jeví se jako zbytečně složitý, vypadá jako slepenec plný těžko identifikovatelných a tedy těžko opravitelných chyb. Navíc informační systém nevykazuje žádné nosné analytické myšlenky a jeho architektura je zbytečně složitá a nepřehledná. Odhalení chyb (například v testování) ještě není zárukou, že se tato chyba skutečně odstraní ve všech místech svého výskytu.
- V některých případech se předchozí bod pojednávající o velmi nízké opětovné použitelnosti v TUNELU rozvine do nejhrůznější podoby: Nejenom, že se začnou opakovat funkcionality v jednotlivých agendách, ale díky opakovaným pracím začne docházet k redundanci samotných evidovaných informací. Například v metodě TUNEL se běžně stane, že některé osoby jsou v systému evidovány několikrát, v každé agendě znovu. Vzniká tak další nečekaný problém: Musí se zadat k řešení úplně „nová agenda“, jakýsi „program pro program“ - zbytečná integrace, která má za úkol dávat dohromady opakující se evidované informace, které se vůbec opakovat nemusely.
- Při metodě TUNEL je každý i malý a jednoduchý analytický požadavek na změnu v konečném důsledku raději odmítán i za cenu obchodních ztrát. Systém vykazuje vysokou „synergetickou nestabilitu“: I malé změny v systému

vedou k jeho nečekaným kolapsům paradoxně v odlehlých částech systému. Díky nízké transparenci, nestabilitě a chybovosti systém vykazuje vysokou pracnost i v případě jednoduché údržby, a to i po drobném zásahu. Po malé změně v systému se vyžaduje obrovský díl práce na opětovném bezchybném stabilním zprovoznění systému (nestabilita jako malá změna vyvolává obrovské nepříjemné důsledky).

- V softwaru se objevují příliš časté chyby i po odevzdávce odběrateli, systém jako produkt vykazuje velmi nízkou kvalitu. Díky chaotickému vývoji a neustálému řešení dalších nových a nečekaných problémů vznikají zákonitě softwarové slepence plné chyb. Důsledkem chybovosti je velmi nízká kvalita produktů. Enormně vzrůstá nutnost neustálých oprav i po implementaci systému. Co je však nejhorší, u zákazníka se začne projevovat nespokojenost s kvalitou dodávky, zejména pokud zjišťuje fatální chyby ve funkcionalitách systému („…ten systém dělá neuvěřitelné hlouposti, takhle jsme to tedy ale vůbec nechtěli…“)
- Chaotický vývoj vede i k zvláštní atmosféře ve firmě, která je charakterizována vysokou hektičností prací, shonem a všeobecnou nervozitou. Vyrobený software se neustále předělává a to stále dokola „jako za trest“. Provádí se zbytečná a tedy úmorně nepříjemná práce. Výsledkem jsou extrémně náročné vztahy na pracovišti. Firma si v konečném důsledku díky neustálým úpravám a opravám uzurpuje velmi vysoké nároky na volný čas pracovníků. Problém nespočívá ani tak v tom, že by se pracovníci bránili příliš vysokému vypjetí sil a přesčasům (pokud mají slušný výdělek), ale jako vysoce inteligentním pracovníkům a expertům jim velmi vadí zbytečnost vykonané práce. Nakonec to mohou pociťovat i jako určitou křivdu, že díky neuvěřitelnému chaosu ve vývoji musejí zůstat přesčas a dané chyby stále dokola opravovat s povzdechem: „Po kolikáté už v této agendě, to už opravdu nikdo neřekne, jak to má být správně?!“ Ve firmě, která pracuje metodou TUNEL, je úplnou samozřejmostí pracovat hodně přesčas, dokonce až tak, že se to považuje za obvyklý standard chování. Někdy se tím vedoucí pracovníci u zákazníků chlubí: „Sledujte, jak usilovně pracujeme, žádné dovolené, zůstáváme tu až do večera, někdy do rána, samé pracovní soboty a dokonce i neděle!“. Ve skutečnosti však firma paradoxně staví na odiv svůj neefektivní způsob práce, který je pro metodu TUNEL příznačný. Správnější než tyto chlubné věty by byla formulace: „Pracujeme sice hodně, ale neefektivně.“ Navíc metoda TUNEL v žádném případě neprospívá dobrým vztahům na pracovišti.
- Souhrnem všech negativních projevů jsou zvýšené náklady a následně finanční ztráty.

Tento způsob tvorby informačních systémů se z uvedených důvodů zásadně nedoporučuje, i když mohu z dlouholeté praxe konzultanta potvrdit, že bývá ve velmi mnoha firmách až příliš často používán.

1.3 Metoda tvorby softwaru byrokratickým způsobem

Druhým extrémem je snaha zbavit se předešlých nedostatků byrokratickým způsobem. Ve zmíněné firmě jsem byl svědkem zavedení této drastické metody.

Jednomu z výše postavených vedoucích se přestala zamlouvat v té době intenzivně působící metoda TUNEL a zejména její důsledky. Rozhodl se tedy s tímto postupem ve firmě rázně skoncovat. Na počátku vcelku správně pochopil, že základním nedostatkem jsou chybějící postupky a chtěl je proto ve firmě zavést. Vzniklo tzv. „Oddělení kvality“, které mělo za úkol vytvořit postupy tvorby softwaru, neboli metodiky. Protože však pracovníci zavádějící tyto postupky v té době neznali základní principy metodik tvorby softwaru, začaly vznikat velmi podivné dokumenty. Podle nich se sice vyžadovalo pracovat, ale pro programátory se stávaly spíše přítěží než pomůckou. Vznikaly tak předpisy podle hesla: „Hlavně ať jsou předpisy!“, avšak jejich správnost, efektivita apod. nebyla posuzována. Doslova začaly vznikat „dokumenty pro dokumenty“.

Tento extrémně opačný „byrokratický“ přístup je charakterizován následujícími body:

- Zavádějí se postupky a metodiky bez zkušeností s nimi, nedodržují se základní pravidla metodik platné pro tvorbu softwaru (o těchto principech bude dále v této e-knize pojednáno)
- Na počátku zavádění metodik se zahajují práce s přehnaným optimismem. Jsou patrná přílišná očekávání ze zavedených postupů, a to s očekáváním výsledků ihned a okamžitě. To samozřejmě vede k zavádění neověřených postupek za každou cenu s následnými krachy.
- Při tvorbě metodik se dopředu předjímají a vymýšlejí nesmyslné postupy neověřené praxí. Metodiky se vydávají jako jednou konečné pravdy, o kterých není již radno diskutovat.
- Přílišné očekávání vede ke snaze o revoluční skoky („čínská revoluce“) ve změnách způsobů práce ve firmě. Zanedbává se ten prostý fakt, že každý fungující mechanismus má svou setrvačnost. Navíc při zavádění metodik je třeba vytvořit rychlou zpětnou vazbu. Každý nový postup vyžaduje „přejít postupně do krve“. Každou postupku je třeba ověřit, odzkoušet, zrevidovat praxí. Při zanedbání těchto faktů následují při zavádění metodik v SW firmách krachy. Správně uvažujícímu člověku by měl přeběhnout mráz po zádech, když někdo přijde s myšlenkou: „Máme tři čtvrtě roku na zavedení postupek a návodek, času máme dost. Zavedeme je a od 1.1. příštího roku je spustíme, to bude potom bomba!“ Bomba to bude, ale jinak, než původně autor

zamýšlel. Necelý rok uplyne a vydají se takové postupky, které všichni zavrhnou jako nesmysly. S příchodem nového roku hora porodí myš.

- Při zavádění byrokratické mašinérie následuje zpomalení vývoje a poté se zvyšuje netrpělivost u zákazníka. V některých případech dokonce pracovníci začnou „švejkovat“ v tom smyslu, že pracují „přesně podle nesmyslných postupek“, což pochopitelně vede ke katastrofálním výsledkům. Důvod je prostý: Pracovníci nemají jinou šanci, jak ukázat nesmyslnost předpisů, podle kterých mají tvořit, než tak, že je přesně dodržují.
- V důsledku tato metoda také vede ke zvýšeným nákladům a ztrátám

Tento přistup k tvorbě softwaru se také nedoporučuje.

1.4 Efektivní technologie tvorby SW, jejich charakteristika a vztah k modelování

Obě předešlé metodiky, jak metoda TUNEL, tak byrokratická metoda, názorně ukazují, jak by metody tvorby softwaru neměly vypadat. Přesto však jsou obě dvě metody v mnoha firmách velmi často používány, zejména metoda TUNEL je až příliš častá. Dokonce se jedná o natolik běžnou praxi, že si pracovníci z těchto firem ani nedovedou představit, že by se mohl anebo dokonce měl software tvořit jinak.

Posouzení obou zmíněných metodik vede ke dvěma zajímavým závěrům:

- První způsob tvorby softwaru zvaný metoda TUNEL ukazuje, kam vede cesta bez metodik, bez postupek a bez návodek. Dokazuje, že je nutné postupky tvorby SW zavést, tj. že je nutné zavést „výrobní linku softwaru řízenou postupkami“.
- Druhý způsob nazvaný jako „byrokratická metoda“ poukazuje na jednoduchý fakt, že pouze splnění předešlého požadavku na zavedení jakýchkoliv metodik, tj. postupek a návodek, není pro získání efektivní tvorby dostatečný. Byrokratická metoda zavádí metodiky dokonce velmi přísně, ale kýžený efekt to nepřináší. Nestačí tedy vyslovit požadavek na tvorbu návodek a metodik, ale je třeba také vědět, jak aby měly tyto metodiky vypadat a jakými principy by se měly řídit.

Uvedené dvě metodiky nám dávají vědět velmi přesně „co nechceme“. Otázkou je, „co tedy chceme“, tj. jak mají správné efektivní metodiky vypadat. Cílem této knihy je mimo jiné ukázat zásady opravdu efektivního způsobu tvorby SW.

Moderní a „správné“ metodiky se vyznačují zejména tím, že odstraňují uvedené nedostatky obou chybných postupů. Technologie efektivní tvorby SW se oproti předešlým výčtům vyznačuje těmito body:

- Správná metodika vykazuje extrémně vysokou efektivitu vývoje při zachování požadavků na dokumentaci v projektu
- Správná metodika je jednoduchá, má vysoký stupeň logiky, transparence, má své jednoduché a lehko aplikovatelné principy.
- Správná metodika je obecná, tj. je použitelná pro široké spektrum vývojových prostředí.
- Správná metodika se vyznačuje jednoduchým a snadným zavedením ve firmách

Když se začneme blíže zabývat otázkou, co vlastně moderní metodiky vedoucí k efektivní tvorbě SW vyžadují, zjistíme jeden zajímavý fakt: Všechny „dobré metodiky“ používají vždy nějakou modelovací techniku softwaru, tj. používají modelování jako svou nedílnou součást. Znamená to, že nezastupitelným prvkem všech metodik je modelování SW. K tomu přistupuje ještě druhý zajímavý poznatek: Protože jazyk UML (Unified Modeling Language) je všeobecně přijímaný standard pro modelování systémů, je valná většina modelovacích technik opřena převážně o notaci UML.

Pokud bychom chtěli odpovědět na otázku z nadpisu podkapitoly „Proč modelovat?“, stačilo by jednoduše říci, že modelování patří mezi postupy, které používají všechny dobré metodiky tvorby softwaru a žádná se bez modelování neobejde. To je sice pravda, ale je velmi důležité pochopit, proč tomu tak je.

Pokud bychom se naučili velmi dobře modelovat a přitom nepochopili základní myšlenku „proč vlastně modelujeme“, můžeme i při modelování sklouznout opět mezi neefektivní postupy, kdy se tvoří spousta zbytečných modelů, ale efektivita vývoje se ztrácí. V takovém projektu se sice vyskytuje spousta modelů, ale žádný z nich není pro vývoj SW zajímavý (modely pro modely).

V další kapitole je proto velmi podrobně vysvětleno, jaký má modelování v efektivních postupech tvorby SW cíl a jak jej proto používat.

1.5 Sémantická mezera

V literatuře se lze dočíst, že za neutěšený stav, který je popsán v metodě TUNEL, je mimo jiné zodpovědná tzv. sémantická mezera. Pochopení významu sémantické mezery vede k prvnímu kroku při nahrazení metody TUNEL efektivními metodami tvorby SW.

Softwarová firma může mít spoustu velmi dobrých odborníků - programátorů a technologů, může mít zkušené „javisty“, „dotnetisty“, „delphisty“ atd., může disponovat zkušenými experty na databáze, ORACLE, MS SQL, experty na XML, JSP, ASP, WWW, XHTML a jiné zkratky, ale pokud nevyřeší problém sémantické mezery, spadne do černého tunelu zmíněné katastrofické metody.

Takže naskýtá se otázka, v čem je problém, když vysoká technologická odbornost pracovníků nestačí?

Podstatou jevu sémantické mezery, jak sám název napovídá, je rozdíl náhledu na vyvíjený software mezi uživatelem softwaru a programátorem tvořícím kód. Následně z toho plyne odlišnost ve vyjadřování (tj. v sémantice) uživatele a programátora. Zatímco programátor vidí svůj výtvor jako souhrn zdrojových kódů, které jsou vysoce specifické po odborné stránce, které řeší problematiku realizace softwaru v daném vývojovém jazyce, uživatel vidí software úplně jinak: Z jeho pohledu se jedná o informační systém, který něco dělá, něco eviduje, něco nabízí, něco provádí a je k nějakému užitku. Jinak řečeno, uživatel a programátor používají při popisu téhož IS „jiné jazyky“. Zatímco uživatel používá svůj vlastní „laický jazyk“, kterým popisuje daný informační systém, programátor používá složitý jazyk daného vývojového prostředí. Protože však oba popisují tentýž informační systém, měli by se takříkajíc v těchto popisech „shodnout“.

A zde je právě největší háček: Základní problém sémantické mezery spočívá v otázce, jak zjistit uvedenou shodu nebo neshodu dřív, než uživatel uvidí hotový software a prohlásí, že to není ono, co chtěl.

Jev sémantické mezery bychom mohli přirovnat k situaci, kdy máme dva texty, jeden v angličtině, druhý ve francouzštině, přičemž oba dva vyjadřují tentýž kuchařský recept. Problém nastane tehdy, když nemáme žádné prostředky překladu, tj. není po ruce nikdo, kdo by posoudil jejich shodu. Efekt sémantické mezery vede k tomu, že musíme zjišťovat shodu obou textů až podle toho, co se uvaří, tj. v našem pojetí až podle toho, co se naprogramuje.

Z této jednoduché úvahy vyplývá jeden velmi důležitý fakt: Proces „překlenutí sémantické mezery“ by měl probíhat již při samotném vývoji jako jeho přirozená

součást a nikoliv „ex post“ až po vyhotovení funkčního kódu, kdy je už skutečně pozdě a „horká kaše je již uvařena“.

1.6 Úrovně abstrakce tvorby SW

Na první pohled by se mohlo zdát, že překlenutí sémantické mezery nemusí být až tak složitý problém. Pokud se vrátíme k přirovnání k textům kuchařských receptů ve francouzštině a angličtině, mohlo by se zdát, že se stačí jen naučit „cizí jazyk“ a problém je vyřešen. Situace je však trochu složitější.

Sémantickou mezeru jsme zavedli jako myšlenkovou propast mezi náhledem uživatele IS a pohledem programátora, který tento IS tvoří. Tím jsme sice jednoduše vystihli podstatu problému, ale tento jev má hlubší povahu. Problém totiž není v samotném efektu sémantické mezery, ale v jeho příčině. Jev, který způsobuje vznik sémantické mezery, se nazývá „úrovně abstrakce tvorby SW“.

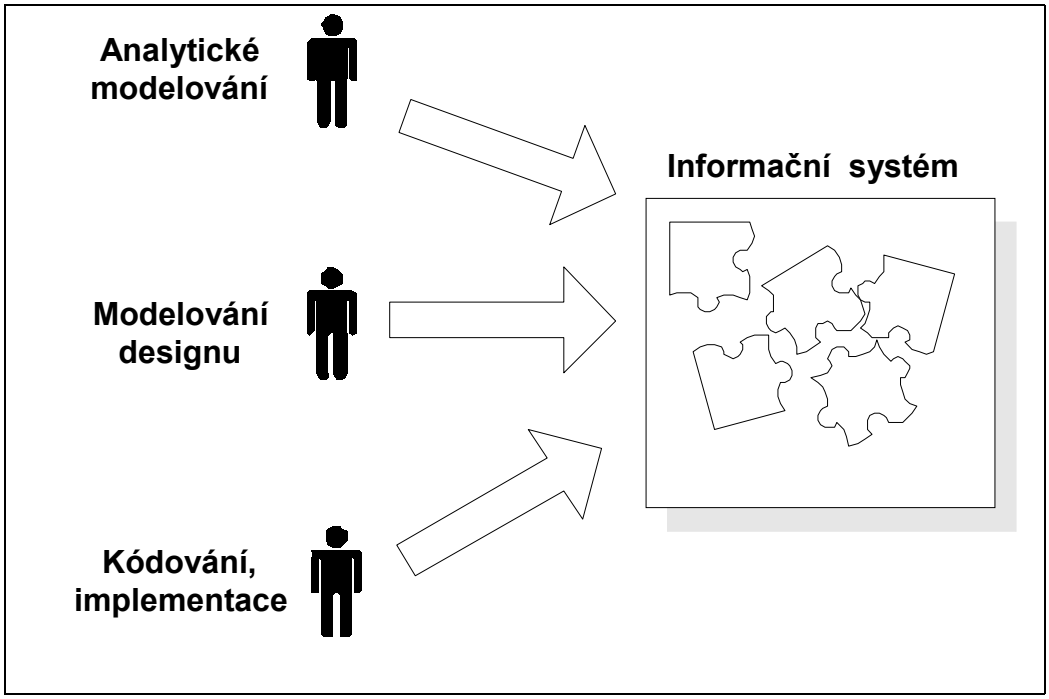
Úrovně abstrakce SW je třeba velmi dobře vysvětlit. Tam se skrývá pravé jádro všech neskonalých problémů při tvorbě SW a v něm se také skrývá možná cesta vedoucí z tunelu. Není divu, že jedním z hlavních pilířů technologie efektivní tvorby SW je důsledné používání tzv. úrovní abstrakce softwaru. Pojem úroveň abstrakce je zaveden ve všech kvalitních metodikách a pro modelování informačního systému se považuje za nezbytný. Správné pochopení úrovní abstrakce usnadňuje nejenom další práce v modelování pomocí UML, ale vede také ke správným postupům ve fázování vývoje informačního systému. Mohu z vlastní zkušenosti potvrdit, že bez zavedení tohoto pojmu se opravdu TUNEL nedá opustit.

Co je to tedy úroveň abstrakce informačního systému? Úroveň abstrakce znamená určitý náhled, ve kterém je do určité míry popis systému poplatný implementačním podrobnostem. Čím více se jedná o nižší úroveň, tím je vyšší detailnost implementačních podrobností a naopak. Dva zmíněné pohledy - programátorský pohled a pohled uživatelský - patří do různých úrovní abstrakce tvorby SW a proto vzniká zmíněná sémantická mezera.

Představme si nějaký informační systém. Na tento systém lze nahlížet z různých úrovní abstrakce, jakoby z různých pater abstrakce. Z hlediska praxe a efektivity technologie tvorby SW se doporučuje rozeznávat minimálně tři úrovně abstrakce:

- analytické modelování (AM)
- modelování designu (D)
- kódování (C)

(viz následující *obrázek 1 Úrovně abstrakce informačního systému*)



obrázek 1 Úrovně abstrakce informačního systému

Důležité je, že i když každý z těchto tří pohledů představuje různá patra abstrakce, tak každý z nich vidí a popisuje jeden a tentýž informační systém. Předmět jejich popisu je stejný. Z toho vyplývá, že přestože se jedná o popisy z různých úrovní, musí se všichni v něčem shodnout. Je vcelku zřejmé, že každý z těchto pohledů používá jiná slova, tj. jinou sémantiku.

Vznikají tak tři typy artefaktů (tj. tři typy výsledků práce), na každé úrovni abstrakce jiné povahy, přitom všechny popisy se týkají téhož informačního systému.

Vysvětlení předešlého obrázku začneme nejnižším pohledem (pozor, nejnižší zde není hanlivý výraz, ale nejnižší z hlediska implementace), který je nejbližší programátorovi.

1.7 Nejnižší úroveň abstrakce - kódování

Na nejnižší úrovni abstrakce je na informační systém nahlíženo pouze jako na kód, který lze zkompilovat do „chodícího systému“. Jedná se o pohled programátora, který tento kód tvoří. Míní se tím nejenom veškerý kód v daném jazyce, ale i SQL příkazy, skripty, zkompilované komponenty apod. Pojem nejnižší úroveň zde nemá hanlivý význam. Je tím myšleno „technologicky nejníže, nejblíž k implementaci“. Pokud se tedy prohlíží zdrojový kód, jedná se o nejnižší úroveň abstrakce daného informačního systému.

Tato nejnižší úroveň abstrakce se také nazývá „implementační úroveň“, „úroveň realizace“, nebo prostě ***kódování*** (v této knize označován dále také zkratka C jako „Code“). Výslednými dokumenty, tj. artefakty, které ztvárňují myšlenky z této úrovně abstrakce, jsou zdrojové kódy, zkompilované balíky souborů, SQL skripty, apod. Spadá sem vše, co v konečném důsledku opravdu fyzicky realizuje a fyzicky instaluje informační systém. Jazyk na této úrovni je složen z rezervovaných slov daného prostředí.

1.8 Nejvyšší úroveň abstrakce - analytické modelování

Protipólem proti nejnižší úrovni abstrakce informačního systému je nejvyšší úroveň abstrakce, která se nazývá ***analytické modelování*** (dále také AM jako „Analytical Modeling“). Výsledkem analytického modelování jsou dokumenty ve formě modelů, které popisují podrobně informační systém v nejvyšší možné abstrakci pouze pomocí pojmů bez jakýchkoliv implementačních podrobností.

Na této úrovni se velmi přesně a podrobně (až do detailů) popíše, jaké pojmy se evidují, jak se tyto evidované pojmy skládají (tj. popíše se struktura evidovaných pojmů), jak se chovají výskyty z těchto pojmů atd. Na úrovni analytického modelování se tedy používají pouze „evidované pojmy“, „výskyty z těchto pojmů“, „chování těchto výskytů z pojmů“ apod. Nikdy se nepoužívají výrazy z nižší úrovně abstrakce z dané technologie, ale pouze výrazy pojednávající o evidovaných pojmech a jejich výskytech.

Je dobré si uvědomit, že toto nazírání, i když zní složitě, odpovídá „zdravému selskému rozumu laika“, který se takto (aniž by to věděl) vyjadřuje. Například hovoří o tom, že „systém bude evidovat běžné účty“ (zavedl evidovaný pojem), nebo prohlásí, že „každý běžný účet patří ke klientovi“ (zavedl interakci - strukturu mezi dvěma pojmy), případně se vyjádří, že „vklad na účet probíhá tak, že …“ (popisuje chování výskytů z pojmů).

Na úrovni AM se pohybují všichni účastníci projektu včetně „laiků“ ve smyslu programování, kteří nazírají na informační systém jako na souhrn evidovaných pojmů a jako na souhrn chování výskytů těchto pojmů. Z pohledu této úrovně abstrakce systém „něco provádí“, „něco umí“, „něco eviduje“, „je k nějakému užitku“ apod. Aniž by se na této úrovni zaváděly jednotlivé elementy z programovacího jazyka, popisuje se podrobně, co systém provádí, jaké informace se evidují a jakou mají tyto informace skladbu. Používají se pouze pojmy jako synonymum pro „evidované informace“ (například faktura, běžný účet apod.). Popisuje se také chování výskytů z těchto pojmů.

Znamená to, že popis systému na úrovni abstrakce AM je oproštěn od prvků a syntaxe daného programovacího jazyka a je ve svém vyjádření implementačně nezávislý. Myšlenky ztvárněné na této úrovni mají povahu modelů informačního systému. Efektivním a standardním jazykem pro jejich zápis je modelovací jazyk UML (Unified Modeling Language). Znamená to, že i tato úroveň má svou přesnou syntaxi, kterou je třeba dobře znát. To je obrovská výhoda: Omezená a přesná syntaxe je dobrým vodítkem k tomu, aby nevznikaly „podivné“ dokumenty AM, které nemají hlavu ani patu.

Důležité je nyní upozornit na skutečnost, že předmětem AM je samotný informační systém a nic jiného. Znamená to, že se jedná o nejvyšší abstrakci napsaného programu, přičemž samotný program je realizován na nejnižší úrovni - kódování. Modely AM nejsou nějakými „obecnými“ větami, ale konkrétním popisem programu, i když úroveň abstrakce je vysoká.

Pomocí modelů v UML se na úrovni abstrakce AM odpovídá na základní otázku „CO?“, tj. odpovídá se na otázku „o co v informačním systému jde, co se eviduje a jak se výskyty informací chovají“. Na této úrovni se pohybují všichni účastníci projektu, včetně těch, kteří neumějí programovat, ale také těch, kteří programovat umějí. V okamžiku, kdy se vyžaduje takzvaně „vysvětlit“ k čemu systém slouží, co eviduje a jak eviduje apod., automaticky se pohybujeme na této nejvyšší úrovni abstrakce.

Úroveň abstrakce AM je používána všemi účastníky projektu včetně uživatelů resp. expertů na danou problémovou doménu, tj. z hlediska tvorby informačních systémů „laiků“. Výstupy z této úrovně jsou po určitých drobných úpravách srozumitelné každému. Díky této skutečnosti se mohou k informačnímu systému vyjadřovat další účastníci projektu, kteří jinak netvoří „programátorský“ tým, což jsou například externí konzultanti, uživatelé, obchodníci apod.

Není bez zajímavosti, že na této úrovni se v určité chvíli pohybuji také programátoři, a to ve chvíli, kdy si potřebují navzájem sdělit „co vlastně programují“. Představme si, že jeden programátor musí předat svou práci jinému programátorovi. Nejprve mu musí tzv. vysvětlit „o co jde “, teprve potom se vysvětluje samotný kód. Programátor by neměl začít vysvětlovat kolegovi, který program vidí poprvé v životě, slovy: „Tady alokuji paměť“. Nejprve začne slovy (například): „Program podporuje evidenci Pokladny v bance…“ (což je věta z úrovně abstrakce AM).

1.9 Střední úroveň abstrakce: Modelování návrhu

Mezi úrovní abstrakce AM a C existuje ještě jedna úroveň, která se nazývá ***modelování designu*** (dále také zkratka D). Synonymem pro tuto úroveň je také název „modelování návrhu“ nebo také pouze „návrh“ resp. „design“. Jedná se o úroveň abstrakce, která je ještě stále „hodně abstraktní“ v tom smyslu, že se nejedná o samotný chodící systém. Znamená to, že se stále ještě jedná o modely, ale tyto modely jsou již technologickým návrhem v daném prostředí. Jinak řečeno na této úrovni vzniká to, co se mnohdy nazývá „technologický návrh“. Jsou to modely poplatné danému prostředí navrhující (zadávající) realizaci v daném vývojovém prostředí. Patří sem například návrh relační databáze, návrh obrazovek, model tříd pro daný OOP jazyk, komponentní model v daném prostředí atd.

Modely návrhu jsou sice „abstrakcí“, ale konkrétním zadáním pro kódování. Modely D „přetavují“ modely z úrovně AM, která je implementačně nezávislá, do konkrétního návrhu (modelu), který je již implementačně závislý. Je třeba zdůraznit, že se stále jedná o návrh, tj. o model. Prvky designu tedy ještě nejsou konkrétními již realizovanými prvky (kód), ale pouze jejich technologickým návrhem. Modelování návrhu se stává zadáním pro nejnižší úroveň abstrakce, tj. pro kódování.

Všimněme si jednoho zajímavého problému spojeného s otázkou „co je to vlastně dobré zadání pro programátora“. Jedním z charakteristických průvodních jevů průchodu TUNELEM je volání pracovníků: „Dejte nám dobré zadání, potom jsme schopni systém dobře realizovat!“ Tento zoufalý výkřik programátorů ve tmě TUNELU je vlastně důsledkem zanedbání existence úrovní abstrakce. Všimněme si, že každá vyšší úroveň abstrakce se stává zadáním pro nižší úroveň abstrakce. Artefakty úrovně abstrakce analytického modelování jsou úplným zadáním pro tvorbu modelů návrhu D a následně modely návrhu jsou úplným zadáním pro kódování C. Například artefakty návrhu D jsou sice stále ještě modely, ale v daném vývojovém prostředí již konkrétně zadávajícími, jak se bude systém realizovat v kódování C. Pro různá prostředí se prvky tohoto návrhu pochopitelně liší podle použité technologie.

Návrh tabulek resp. tříd objektového programování, funkcí, návrh souborů apod. odpovídá vždy danému prostředí a pro jiné prostředí bude tento návrh designu jiný.

Zatímco úroveň abstrakce AM odpovídá na otázku „CO?“, modelování návrhu D odpovídá na otázku „JAK?“ ve smyslu „jak má být to, co je popsáno v analytickém modelování, navrženo a následně naprogramováno v daném prostředí“.

Modelování návrhu D je na rozdíl od AM již implementačně závislé a poplatné danému prostředí. Na rozdíl od úrovně AM se na úrovni abstrakce modelování návrhu D nepoužívají výhradně CASE nástroje podporující UML (i když i to je možné). Ukazuje se jako efektivní použít pro tvorbu výstupů z této úrovně ve větší míře nástroje daného vývojového prostředí. V některých případech bývá tvorba některých prvků kódování zjednodušena automatickými procesy (generace kódu apod.). Vznikají tak dokumenty designu jako zadání pro tvorbu kódu.

Ještě jeden fakt je třeba zdůraznit: Ať chceme anebo ne, při tvorbě softwaru zákonitě vždy existují tyto úrovně abstrakce analytického modelování, designu a kódování. Vždy se při vývoji informačního systému těmito úrovněmi prochází, jedinou otázkou je, zda jsou tyto „průchody“ dobře zdokumentovány (nutno podotknout, že většinou bohužel s výjimkou kódu velmi špatně). Je zřejmé, že vždy, tedy i při tvorbě toho nejjednoduššího softwaru, se nejprve zamyslíme „o co tam jde“, potom se zamyslíme „jak toto zrealizovat v daném prostředí“ a následně to kódujeme.

Vrátíme se k problematice sémantické mezery. Jak vidět, sémantická mezera vzniká jako důsledek existence úrovní abstrakce: Uživatel se pohybuje v prostoru, který je vymezen abstraktním modelováním a je na vyšší úrovni abstrakce, kdežto programátor se pohybuje v prostoru kódování, tj. z hlediska implementačních podrobností na velmi „hluboké“ úrovni abstrakce . Z toho důvodu oba vidí systém tak trochu jinak jakoby z jiného úhlu (z různých pater) a vzniká tak sémantická mezera. Efekt „překlenutí sémantické mezery“ není nic jiného, než „dobrý přechod od analytického modelování AM přes design D až do kódování C“.

1.10 Příklad - řeč programátora a řeč uživatele

Jako příklad rozdílného náhledu úrovní abstrakce uvedu klasický dotaz jednoho programátora na školení:

„Jak si mám u uživatele, tj. u budoucího odběratele systému, ověřit, že můj napsaný příkaz SELECT z databáze je opravdu v pořádku? To mu mám vysvětlit syntaxi SQL?“

Odpověď zní:

„V žádném případě, to nemůžete. Je třeba uvedený ‘dotaz do databáze‘ vyjádřit na úrovni analytického modelování, kterému bude uživatel rozumět.“

Samozřejmě „jak se to dělá“ je mimo jiné předmětem této knihy.

1.11 Mapování analýzy do designu a následné kódování

Všechny moderní metodiky tvorby SW nějakým způsobem zavádějí ať už přímo nebo nepřímo podobné abstraktní úrovně, i když mnohdy se nazývají jinak. Například pro úroveň, kterou jsme zde nazvali jako „analytické modelování“, můžeme v různých metodikách včetně doporučených postupů návrhů databází apod. najít blízké pojmy, které se liší určitou syntaxí, odlišují se objektovou čistotou apod. Vždy se však snaží oddělit od sebe jednotlivé úrovně abstrakce tvorby SW. Příkladem jsou fáze vývoje z těchto metodik s názvy jako „analýza a návrh“, „tvorba logického modelu“, „tvorba konceptuálního modelu“, zavedení pojmu „entita“ apod. Jak vidět, u všech těchto postupů je (ať už s větším nebo menším úspěchem) patrná snaha o oddělení dokumentace z různých úrovní abstrakce. Určité neúspěchy a mnohdy zbytečná složitost některých metodik je dána nízkou úrovní objektově orientovaného přístupu.

Kvalitativním vyvrcholením těchto snah je zavedení jazyka UML pro modelování analytických dokumentů případně dokumentů designu. Výhodou UML je kromě jiného to, že je speciálně určeno pro objektově orientovaná prostředí a proto je pro ně velmi silným modelovacím jazykem.

V každém případě je vidět, že zavedení úrovní abstrakce není jakýmsi specifikem nějaké školy modelování, resp. že se nejedná o něčí geniální výmysl. Existence úrovní abstrakce je objektivní skutečností. Jak bylo řečeno, tyto úrovně abstrakce a následně fáze vývoje při tvorbě SW existují vždy, ať už chceme, nebo nikoliv. Každý software, který je třeba vytvořit, musí zákonitě projít zmíněnými úrovněmi abstrakce AM + D + C. Tomuto postupu přechodu od vyšší úrovně abstrakce k nižší se nelze nikdy vyhnout a provede se vždy a za každých okolností, pouze mnohdy nebývají tyto fáze zdokumentovány.

Přechodu od analytického modelování do designu budeme také říkat mapování do designu. Problém překlenutí sémantické mezery jsme označili jako dobře učiněný postup průchodu úrovněmi abstrakce od analytického modelování přes design až po kód, tedy jedná se o dobře učiněné mapování z AM do D s následnou realizací v C.

Metoda TUNEL v zásadě nezohledňuje jednoduchou skutečnost existence AM + D + C úrovní a průchod těmito úrovněmi. Jako výsledek se požaduje „pouze kód“, případně možná „něco málo z designu“ (např. diagram databáze apod.). Analytické modely a mnohdy dokonce i modely designu sice existují v myslích autorů (někdo se

musel zamyslet, co a jak se vlastně programuje), ale samotná dokumentace z těchto úrovní abstrakce není vytvořena. Myšlenky z vyšších úrovní abstrakce zůstávají pouze v hlavách tvůrců.

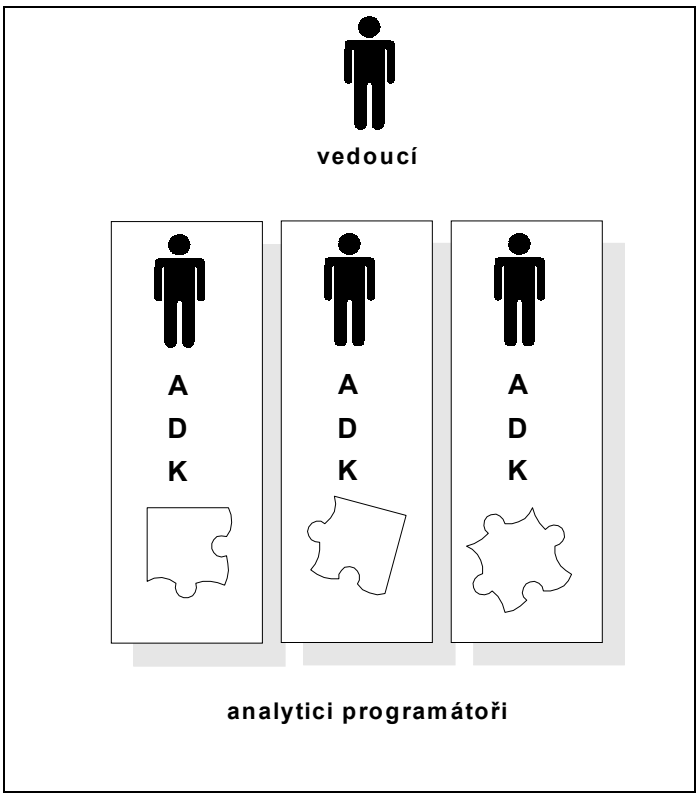
Je zřejmé, že pokud se požaduje zdokumentovat vývoj ve své úplnosti, musí být dostatečně zdokumentován na všech třech úrovních AM + D + C. Bohužel mohu z několikaleté praxe konzultanta potvrdit, že mnoho SW firem se dopouští té chyby, že nemá dostatečně zdokumentovánu úroveň abstrakce analytického modelování a také modelování designu je mnohdy velmi slabé. Tímto vlastně tyto firmy zákonitě padají katastrofických scénářů metody TUNEL.

1.12 Metoda příčného řezu

Jedna z hlavních příčin, proč firma samovolně dospěje k metodě TUNEL, spočívá hlavně v chybné metodě řízení projektů. Vymezení fází analytického modelování, designu a kódování přesně určuje, jaké dokumenty mají kdy vzniknout a v jakém jsou vztahu. Z tohoto hlediska se jeví jako velmi užitečné, aby analytické práce prováděl určený pracovník v roli analytika a modelování návrhu prováděl pracovník v roli designéra. První z nich zná velmi dobře problémovou doménu a je schopen pomocí UML a vhodných nástrojů a postupů modelovat zadání pro designéra.

V mnoha SW firmách se však na rozdíl od oddělení fáze analýzy a designu volí jiný přístup tvorby SW. Nazvěme tento přístup řízení pracovníků jako „*model příčného řezu*“.

Použije se tento jednoduchý postup: Mezi pracovníky (původně programátory) se rozdělí práce tak, že se systém jednoduše rozčlení na části, nazvěme je agendy, a pracovníci je dostanou na starost. Každý z nich potom provádí všechny práce od analytického modelování, přes design až po kódování, každý pracuje pouze na „své části IS“. Rozložení prací v projektu lze znázornit takto:



obrázek 2 Model příčného řezu

Všimněme si, že všichni programátoři mají na starost „svůj výsek“ IS“ a provádějí na něm práce jak analytické, tak práce návrhu designu a kódování. Zvolili jsme název pro tento model „metoda příčného řezu“, protože tvorba IS se dělí nikoliv podle úrovní abstrakce, ale napříč podle oblastí řešení (podle agend).

Například ve zmíněné softwarové firmě vyrábějící bankovní software existovaly agendy jako „Pokladna“, „Termínované vklady“, „Běžné účty“ apod., které se rozdělily mezi pracovníky. Tito pracovníci je tvořili takříkajíc „od A až do Z“. *(poznámka: Dnes se mi doslova pod těmito agendami vybavují bývalí kolegové jako personifikace těchto agend ☺)*.

Tento způsob řízení projektů je ve firmách velmi častý a je oblíben pro svou jednoduchost, přesněji řečeno pro jednoduchost z pozice vedoucího, nikoliv pracovníků. Právě jednoduchost řízení a také jednoduchost zavedení jsou jedinými výhodami této metody. Jako vedoucí se nemusíme nějakým způsobem zajímat o překlenutí sémantické mezery, o abstraktní úrovně apod., protože o přechod z analytického modelování do designu a kódu se musí každý pracovník postarat sám. Každý se s tím vypořádá podle svého, protože to tak má zadáno. Pro vedoucího tedy

stačí „prostě rozdělit práci“ a potom kontrolovat, jak se pracovníci „snaží podat výkony“.

Přestože se jedná o velmi rozšířený model řízení, vřele jej nedoporučuji. Metoda příčného řezu a metoda TUNEL popsaná v předešlých kapitolách spolu úzce souvisejí: Metoda TUNEL je realizována pomocí metody příčného řezu a proto jdou obě spolu „ruka v ruce“. Použití metody příčného řezu může vést k velmi špatným až fatálně chybným výsledkům zejména u větších projektů. Všimněme si těchto hlavních nevýhod této metody a jejích nepříznivých důsledků.

- Předešlý obrázek připomíná spřežení koní a každý kůň má klapky na očích. Programátor pracující na své agendě „neví“ nic o sousedovi. Jednotlivé agendy jsou od sebe odstřiženy, bez opětovné použitelnosti, bez re-use, bez logické návaznosti. Předávání výsledků práce je velmi obtížné a nakonec se provádí pouze ústně a náhodně. *(Poznámka, spíše postřeh: Největší výhodu ve firmě mají kuřáci, kteří si předávají slovně své poznatky na kuřárně u cigarety, mnohdy i nekuřáci se tam dostavují, dýchají pasivně kouř jen proto, aby se něco dozvěděli ☺.)*
- Nejenom, že se opakují práce, ale systém není navíc přehledný, logický a transparentní. Agendy pracují bez vzájemné logiky mezi sebou, chybovost prací je enormní, navíc chyby se obtížně lokalizují, protože věci nejsou na svých logických místech, kde by měly být a opakují se všude možně, „bůh ví kde“.
- Systém nevykazuje žádnou logickou nosnou myšlenku. Jedná se o nesourodý slepenec agend.
- Existuje reálné riziko, že pracovník bude optimalizovat svůj proces tvorby SW tak, že vynechá dokumentaci analýzy a designu. Protože vyvíjí pod enormním časovým tlakem kdy „včera bylo pozdě“, není se čemu divit, že vynechá povinnou část dokumentace, kterou má v hlavě a může ji tedy sám použít. Proč dokumentovat něco, co vím. Výsledkem jeho prací je pouze nepřehledný kód plus možná jakési velmi nepřehledné poznámky o designu. Analýza zůstala pouze jako myšlenky v jeho hlavě. Důsledky jsou pro firmu hrozivé: Pracovník se nejenom že stává nezastupitelným a nenahraditelným, ale navíc mnohdy i nepoužitelným pro další vývoj na jiných pracích, protože nikdo kromě něj nemůže tuto osobu nahradit, a to mnohdy dokonce i po implementaci systému a po ukončení hlavních vývojových prací. Nikdo jiný agendě totiž nerozumí. Co je však ještě horší, nikdo jiný si nemá co přečíst, aby jí rozuměl. Vzniká začarovaný kruh: Díky nezastupitelnosti pracovník nestíhá, takže není čas, aby zpětně cokoliv zdokumentoval a odevzdal část své práce kolegovi.

- Vedoucí, který by případně vyžadoval jakékoliv dokumenty z AM nebo D, vypadá jako ten, který zdržuje. Při metodě příčného řezu se těžko zavádějí jiné metodiky než TUNEL.
- Pracovník, který se stane nenahraditelným, má sice dočasnou výhodu vůči firmě, ale nakonec na to doplácí nejvíce on sám. Nejenže jakékoliv dohody ohledně volného času, dovolených apod., narážejí na jeho nezastupitelnost, ale ještě ke všemu pracovník dostane svou agendu „na doživotí“ a jako ve starých bájích drží „převozníkovo veslo, které nemá komu předat“. Takovýto pracovník většinou časem „zakrní“, technicky neroste a jeho vzdělávání dostává vážné trhliny. Před touto situací případné vaší nezastupitelnosti důrazně varuji.
- Pracovník se stává „všeumělem“. Nejenom že rozumí dobře problémové doméně, například zahraničním akreditivům v bance nebo obchodování s cennými papíry, ale umí i teorii databází, administraci databáze, umí programovací jazyky (například JAVA nebo C#), zná nastavení hardwaru, instalace, umí spravovat webovský server a spoustu jiných věcí. Metoda příčného řezu svým zaměřením stírá specializaci a brzdí tak odborný růst zaměstnanců. Pracovník vyvíjí vše od A až do Z a je tedy specialistou na všechno, což je samo o sobě protimluv. Výsledkem takového postupu je závěr, že ve firmě se nakonec vyskytují pracovníci sice s bohatými znalostmi, ale nikoliv dobří „guru“ a dobří specialisté na nějakou oblast.

Použití metody příčného řezu spolu s metodou TUNEL lze považovat za jeden z nejrozšířenějších nešvarů řízení projektů a jeho odstranění vyžaduje zavedení jiných „sofistikovanějších“ postupů tvorby SW. Je třeba opustit primitivní způsob dělení prací podle schématu ve smyslu agend jako celků.

Moderní metodiky nabízejí jiný způsob dělení prací: Nikoliv napříč přes agendy, ale přes úrovně abstrakce.

1.13 Jak se padá do pasti metody TUNEL a jak se z ní dostat

Předešlý výklad zřetelně názorně ukazuje, jak vlastně vypadá scénář pádu do pasti metody TUNEL. Postup je následující:

- Softwarová firma zahajuje práce na projektu, musí rozdělit práci mezi zaměstnance.
- Aniž by se o daném problému vědělo, musí se vyřešit problém překlenutí sémantické mezery, tj. průchod úrovněmi abstrakce při tvorbě SW. „Někdo“ musí zjistit, co se má programovat, „někdo“ musí navrhnout design (technologický návrh) a „někdo“ to musí naprogramovat. Je možné, že ten „někdo“ bude v rámci dané oblasti řešení jedna osoba a tak se nepřímo zavede metoda příčného řezu.
- Zvolí se nejjednodušší řešení, které je automaticky po ruce: Problém se rozdělí na podoblasti, ty se přiřadí zaměstnancům. Ten „někdo“ je ve všech třech případech programátor. Sám zjistí, co se má programovat, sám si to navrhne a sám naprogramuje. Začíná fungovat metoda příčného řezu a spolu s ní i metoda TUNEL.
- Řízení projektů (dá-li se tak průchod tunelem nazvat) spočívá v nekonečných poradách, neustálých změnách v zadání, roste počet chyb atd. (viz důsledky metody TUNEL a metody příčného řezu).

V některých případech firma setrvává v tomto stavu a „potácí se“ tunelem od projektu k projektu. Někdy se však objeví snaha o nějaké řešení a pokus vymanění se z této pasti. Hrozí však určitá rizika. Další scénář může vypadat takto:

- Nepříznivé důsledky metody příčného řezu a metody TUNEL vedou ke snaze situaci ve firmě řešit.
- Dospěje se k závěru, že je třeba vytvářet analýzu a design, ale neví se jak. Určí se proto někdo jako analytik a ten začne tvořit „jakési“ dokumenty, které však neodpovídají požadavkům na dokumenty analytického modelování. Pro účely designu se v nich objevuje spousta nezajímavých věcí, dokumenty nemají v žádné případě povahu zadání pro další práce designéra a programátora. Práce analytika je svým způsobem velmi neefektivní, v krajním případě až zbytečná. Díky neznalosti problematiky postupů tvorby SW (zejména neznalost jak tvořit dokumenty analytického modelování) nyní hrozí dvě možné pasti, kam firma může v dalším postupu spadnout:

- Protože dokumenty analytika neodpovídají zadání pro design a programování, ale přitom nepřímo obsahují dostatek informací pro vytvoření tohoto zadání, designér a programátor nakonec podle dodaných dokumentů od analytika sami vytvářejí prvky analytického modelování, tj. „sami si ve svých hlavách analyticky namodelují systém“. Tím vlastně suplují práci analytika, který se dostává do pozice spíše konzultanta jako zdroje informací pro modelování systému. V důsledku se nakonec programuje podle jiných dokumentů, než jaké dodal analytik. Výsledkem této snahy je v tomto případě nenápadný až skrytý návrat do metody TUNEL, protože se programuje podle dokumentů, které neexistují (zůstaly pouze v hlavách tvůrců)
- Druhá možnost je, že se ve firmě se určí kategorickým příkazem, že takto podle těchto dokumentů od analytika se musí postupovat, bez ohledu na to, zda jsou nebo nejsou relevantní. Výsledkem je pád do pasti byrokratické metody, kdy se musí postupovat podle nesmyslných dokumentů.

Jak se těmto katastrofickým (a bohužel příliš častým) scénářům vyhnout? Je třeba si zodpovědět tyto základní otázky:

- jak vlastně tvořit dokumenty na jednotlivých úrovních abstrakce, co musí povinně obsahovat každá úroveň a co je „zbytečná omáčka“?
- jak přecházet z jedné úrovně do druhé (mapování) a co přitom zdokumentovat?
- jakým způsobem rozdělovat práci mezi pracovníky při zohlednění předešlých bodů?

V dalších kapitolách je velmi názorně vysvětleno, jak se těmto pastím vyhnout a tvořit SW efektivně, tj. budou velmi podrobně rozebrány odpovědi na předešlé tři otázky

1.14 Příklad na abstraktní úrovně: Převodní příkaz v bankovním informačním systému

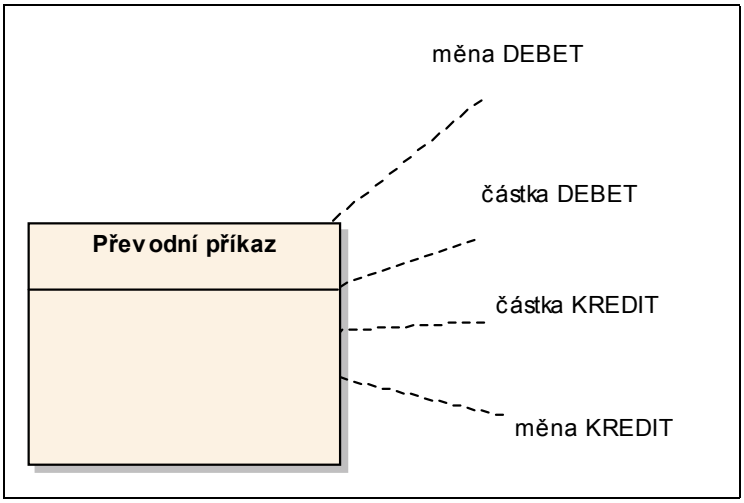
V bankovním informačním systému byl zaveden (a následně naprogramován) tzv. převodní příkaz. Na základě tohoto příkazu dojde v systému k převodu peněz z jednoho účtu na druhý. Tento převodní příkaz byl tzv. víceměnový, tj. účet debet, odkud se peníze braly, mohl být v jiné měně než účet kredit, na který se peníze

připisovaly. Znamenalo to, že v převodním příkazu existovaly dvě částky, částka debet a částka kredit, každá ve své měně. Při převodu peněz se podle aktuálního kurzu přepočítala z částky debet částka kredit do druhé měny.

Jednotlivé operace v systému byly chráněny proti pádu systému a ty příkazy, které se nedaly uskutečnit, spadly do tzv. „koše převodních příkazů“. Tam se daly prohlížet, zjišťovat, proč neproběhly apod.

Všimněme si, že předešlé věty popisují strukturu informace a chování výskytů na úrovni abstrakce AM, nikde se nehovoří o to, „v čem to bylo napsáno“.

Schématicky si znázorněme, jaké informace obsahoval převodní příkaz například takto:



obrázek 3 Skladba převodního příkazu, úvodní poznámky

Tento příklad se stal ještě před zavedením měny euro. Systém běžel v bance bez problémů nějakou dobu, až v jednom okamžiku se v koši objevil převodní příkaz, který nešlo uskutečnit. Začalo se zkoumat proč.

Měna debet tohoto převodního příkazu byla v librách, měna kredit byla v lirách. Navíc se jednalo o velmi vysokou částku (například něco jako vyrovnání mezi národními hospodářstvími apod.). Převodní příkaz spadl do koše na chybu přetečení (tj. „overflow“) částky kredit. Protože mezi měnou libra a měnou lira je několikanásobný řádový posun, navíc sama výchozí částka byla vysoká, došlo k přetečení částky v daném formátu na straně kredit. Daný převodní příkaz spadl do koše jako neuskutečnitelný.

Představme si, že systém nebyl navržen metodou příčného řezu, ale podíleli se na něm pracovníci v rolích analytika, designéra a programátora. Existují tedy dokumenty analytického modelování, designu a existuje zdrojový kód.

Jste vedoucí projektu a ptáte se: Který z těchto tří udělal chybu vedoucí k tomu, že převodní příkaz spadl do koše na chybu přetečení? Je to chyba analytika, designéra nebo programátora? Kde najdeme chybu, ve kterém dokumentu?

Než budete číst další řádky, zkuste si sami v duchu odpovědět!

Není bez zajímavosti, že takřka vždy, když jsem tento příklad uváděl na školeních, tak většinou odpovídali absolventi kurzu na tento dotaz „určitě designér“, ten navrhuje formát částky, někteří se přiklonili k verzi „možná také analytik“, v každém případě programátora jako chybujícího vyloučili. K této zajímavé statistice, která vylučovala programátora, pouze podotknu, že většinu posluchačstva tvořili programátoři.

Správná odpověď zní: Chybu mohl udělat kterýkoliv z nich. Při jejím hledání by se správně mělo postupovat takto: Jděme „odzadu“ ve smyslu následnosti tvorby a to vždy „realizace versus zadání“. Zkontroluje se, zda realizovaný kód odpovídá zadání z designu. Designér například v tabulce navrhl částku v určitém formátu databáze, ale v přímo v „reálné chodící databázi“ u klienta nalezneme jiný formát (nebo nesoulad u formátu proměnné, atributu objektu apod.). V tom případě se programátor dopustil chyby a nesplnil zadání od designéra.

Pokud však nalezneme shodu, tak programátor splnil zadání a „má alibi“. Jdeme dále k následujícímu přechodu mezi úrovněmi. Podobně se musí zkontrolovat, zda pole částky definované v designu odpovídá zadání, které určil analytik ve fázi analytického modelování. Pro někoho se to může zdát jako překvapivé, ale v našem příkladu je formát částky definován již ve fázi analytického modelování! Znamená to, že již zde, na úrovni analytického modelování se definuje, jaký bude mít částka formát. Nepoužije se však žádný existující formát nějakého prostředí (nějaké „money“, „varchar“ apod.), ale tzv. analytický formát. Jedná se o přesně definovaný formát popsaný v obecné formě, například u částky se jedná o popis ve smyslu „číslo s pevnou čárkou, tolik a tolik před čárkou, tolik a tolik za čárkou“, nebo nějak takto „XXXXXXXXXX.XX“ apod. Takovýto formát můžeme nalézt například ve státních normách, které nemohou být napsány specielně pro nějaké vývojové prostředí, ale obecně pro „všechna prostředí“.

Designér dostane takovéto analytické zadání a následně svým návrhem musí toto zadání od analytika splnit. Jinak řečeno, při hledání chyby se musí zkontrolovat shoda mapování z analytického mapování do designu daného prostředí.

Může se stát, že designér skutečně splnil zadání od analytika, jde se tedy za analytikem s dotazem, proč je v analytickém dokumentu navržen formát částky zrovna takto, když nakonec došlo k „přetečení“. Zdá se, že v tomto případě je chyba na straně analytika, protože jsme doputovali až k němu. Může se však stát, že

analytik na náš dotaz odpoví: „Ale v bance mi dali tento dokument ‘Předpisy XYZ v bance‘, a tam je skutečně takovýto formát!“ V tomto případě by byl i analytický návrh částky v pořádku, protože takovýto převodní příkaz s tak vysokou částkou by banka vůbec neměla používat. Zde by se analytikovi dalo vytknout pouze jediné: Systém by měl být v tomto případě navržen tak, aby již při zadávání převodního příkazu byla obsluha upozorněna, že takovýto příkaz by nemusel být v daném převodním kurzu měn přijat.

Uvedený příklad je velmi poučný. Ukazuje, jak se vlastně prochází jednotlivými abstraktními úrovněmi. Všimněme si na něm ještě jedné zajímavé skutečnosti: Jako první navrhuje formát analytik. Designér je expertem na dané vývojové prostředí. V metodě TUNEL a metodě příčného řezu všechno vymýšlí a vše navrhuje jedna osoba. Jak můžeme po expertovi na databáze, specialistovi na programování, na komponentní technologii atd. chtít, aby znal jaké jsou převodní kurzy mezi měnami libra a lira? Musí to být analytik, který od těchto analytických problémů designéra doslova „odbřemení“. Analytik dodá do projektu takové dokumenty, že designér se věnuje již pouze návrhu technologie (např. DOT NET plus MS SQL, nebo JAVA plus ORACLE, nebo prostředí CACHE apod.). Designér by se neměl věnovat problémové doméně, která je pro něj při dobrém analytickém zadání podružná.

S trochou nadsázky se dá říci, že designérovi je při dobrých analytických dokumentech úplně jedno, co vlastně navrhuje, zda „hrušky nebo jablka“. Pokud analytik dodá dobré dokumenty AM (například pomocí UML), dobrý designér je schopen dobře navrhnout jak agendu zahraničních akreditivů, tak docházku do firmy stejně jako evidenci zvířat v zoologické zahradě.

1.15 Požadavek na úplnost dokumentace tvorby SW

Předešlý příklad vede mimo jiné k odpovědi, co vlastně musí obsahovat dokumenty v daných úrovních abstrakce. Z hlediska efektivní technologie tvorby IS je dokumentace AM + D + C natolik úplná, že:

- chodící kód (C) je úplný, je vyhovující ve funkčnosti (bezchybný), transparentní a lze jej opakovaně kontrolovat a upravovat
- modely designu (D) jsou dostatečné jako zadání pro tvorbu kódování a lze je opakovaně kontrolovat a upravovat
- modely úrovně analytického modelování (AM) jsou dostatečné jako zadání pro tvorbu designu a lze je opakovaně kontrolovat a upravovat

Jedním z hlavních cílů technologie efektivní tvorby SW je právě stanovení postupů pro maximálně efektivní vytváření artefaktů na úrovních abstrakce AM+D+C tak, aby tento postup nepřinášel zpoždění v realizaci projektu.

Velmi častou chybou při chápání abstraktní úrovně analytického modelování je nedocenění její poměrně hluboké podrobnosti. Skutečnost, že tato úroveň abstrakce je nejvyšší, neznamená, že se jedná o jakýsi mlhavý, nebo dokonce neurčitý popis. Modely na této úrovni jsou velmi podrobné a obsahují velmi mnoho informací, tj. popisují velmi podrobně informační systém. Modely AM obsahují vše, co je třeba k naprogramování systému, pouze obsah těchto modelů je implementačně nezávislý. Popis systému neobsahuje konkrétní prvky z konkrétního vývojového prostředí. Neobsahuje konkrétní rezervovaná slova z programovacích jazyků a prostředí (jako je například `unit`, `record`, `label`), neobsahují konkrétní prvky obrazovek z prostředí (jako je například `CommandButton`, `TextBox`, `Listbox`), neobsahují syntaxi z konkrétních relačních databází (jako je například "`SELECT * FROM TOSOBA`") apod.

Namísto takovéhoto konkrétního vyjádření v daném prostředí se používají abstraktnější vyjádření, jako jsou věty například: „Obsluze se zobrazí seznam faktur“, „Obsluha vybere..., obsluha zadá...vypočte se… “ apod. Přitom tento popis se chápe jako úplný, takže pokud se něco v modelech analytického modelování nenachází, nebude to naprogramováno.

1.16 Dokumenty typu AM+D+C a dokumenty typu přechodu mezi úrovněmi

Jednou z nejčastějších chyb při vývoji informačního systému je nedodržení nutnosti oddělení jednotlivých úrovní abstrakce od sebe při současném popisu přechodu z jedné úrovně do druhé. Při návrhu systému se v jednom dokumentu chybně použije náhledu současně z několika abstraktních úrovní, tj. provede se jejich „promíchání“. Každý popis informačního systému je vymezen svou úrovní abstrakce, tj. buď patří do AM, nebo patří do D, nebo spadá do C. Tím je také ohraničen ve své abstrakci. Vyžaduje se proto přesně definovat úroveň abstrakce, kam daný dokument spadá a dodržet tak čistotu daného dokumentu.

První základní pravidlo pro správné metodiky zní: Dokumenty z analytického modelování, designu a kódování jsou do sebe striktně odděleny v tom smyslu, že všechny detaily dokumentu AM nebo D nebo C spadají vždy do právě jedné dané abstraktní úrovně AM nebo D nebo C.

Z toho například plyne, že je chybou, pokud například vznikne dokument, který se prohlásí za dokument analytického modelování, ale obsahuje i některé z prvků designu. Jedná se mimochodem o velmi častou chybu.

Navíc se ukazuje, že pro celkovou dokumentaci vývoje nestačí pouze vytvořit artefakty na třech úrovních abstrakce AM+D+C. Pokud se totiž jedna úroveň abstrakce opouští a přechází se na druhou nižší úroveň abstrakce, nemusí se jednat pouze o jeden možný postup tohoto přechodu , tj. jedinou alternativu přechodu. I když se jedná o jedno a to samé technologické prostředí designu, tak při mapování z analytického modelování existuje vždy několik variant, jak splnit zadání analytika a vytvořit tak artefakt designu. Designér má vždy na výběr z několika technologických variant. Proto se musí také tento přechod mezi úrovněmi AM a D náležitě popsat a zdokumentovat.

Z uvedeného vyplývá, že kromě dokumentů tří úrovní abstrakce vznikají ještě navíc dokumenty obsahující popis způsobu přechodu mezi nimi. Požadavek na úplnost dokumentace tedy navíc vede k nutnosti popsat postupy přechodů mezi úrovněmi abstrakce od vyšší k nižší úrovni. Dostáváme tak jeden důležitý závěr:

Celý vývoj informačního systému se tak stává tvorbou dokumentů ze dvou oblastí:

- dokumentace modelů z jednotlivých úrovní abstrakce AM+D+C čistě od sebe oddělených
- dokumenty popisující přechody mezi úrovněmi abstrakce

Teprve až souhrn artefaktů z těchto dvou typů dokumentace dává ucelenou dokumentaci vývoje.

První oblast vymezuje „jak je systém vymyšlen, navržen a nakódován“, druhá oblast vymezuje, „jak se k těmto dokumentům jako plnění zadání vlastně došlo“.

1.17 Fázování prací vývoje SW

Tři různé pohledy na IS z hlediska abstraktních úrovní AM+D+C implikují posloupnost prací na vyvíjeném IS, tj. ***fázování projektu***. Postupuje se vždy logicky od nejvyšší úrovně směrem k nižší úrovni abstrakce, tedy od abstrakce analytického modelování AM do designu D a z designu D do kódování C.

Je dobré si připomenout jeden velmi důležitý fakt: Každý tvořený software vždy projde těmito úrovněmi abstrakce, ať si to přejeme anebo nikoliv. Vždy, i při nejjednodušším programu, se musíme zamyslet nad tím, „o co tam jde“ (fáze analytického modelování), „jak toto navrhnout v daném prostředí“ (fáze designu) a poté nakódovat (zrealizovat). V některých případech první dvě fáze AM a D proběhnou ve zlomku vteřiny a jde se programovat. Znamená to, že mnohdy (a někdy až příliš často) nebývají fáze analytického modelování AM a designu D

dostatečně zdokumentovány, což neznamená, že neexistují. Existují, ale pouze jenom jako myšlenky v hlavách autorů.

Časová posloupnost tvorby z analytického modelování AM do D a následně do kódování C však neznamená, že musí být hotovy všechny prvky AM v úplnosti k tomu, aby se mohlo přistoupit k tvorbě modelování designu D. Zákonitě vždy však platí, že:

- na základě modelů analytického modelování vznikají dokumenty modelování návrhu
- na základě výsledku modelování designu vznikají následně dokumenty zdrojového kódu jako je samotný zdrojový kód, SQL příkazy apod.

Výsledek vyšší úrovně abstrakce se stává zadáním pro tvorbu nižší úrovně abstrakce. Postup mapování z analytického modelování AM do designu D je také nutnou součástí dokumentace.

1.18 Efektivní přechod z analytického modelování do designu a do kódu

Při zavádění moderního způsobu tvorby informačního systému při respektování úrovní abstrakce je mnohdy jednou z často kladených otázek, zda tvorba modelů na úrovních AM+D+C příliš nezpomalí celkový vývoj. Je třeba podotknout, že při nesprávném přístupu opravdu takovéto riziko hrozí. Firma se může utopit v horních úrovních abstrakce, aniž by byl přitom viděn cíl, tj. naprogramovaný systém. Pracovníci neustále malují zbytečné obrázky, přičemž efekt z hlediska programování není buď žádný anebo je minimální. Nakonec se paradoxně programuje podle jiných dokumentů resp. podle ústního zadání.

Jedním z přínosů efektivní technologie tvorby IS jsou pracovní postupy, které odstraňují toto riziko a vedou skutečně k extrémně rychlé tvorbě informačního systému při úplnosti dokumentace na všech úrovních abstrakce. Řešení spočívá ve dvou vzájemně souvisejících bodech:

- zavedení správné, přesné a efektivní syntaxe modelování na horní úrovni abstrakce AM, tj. syntaxe abstraktního programování v AM využívající UML
- přesně definované a opakující se postupy mapování z vyšší úrovně abstrakce do nižší, tj. zavedení vzorů mapování z AM do D

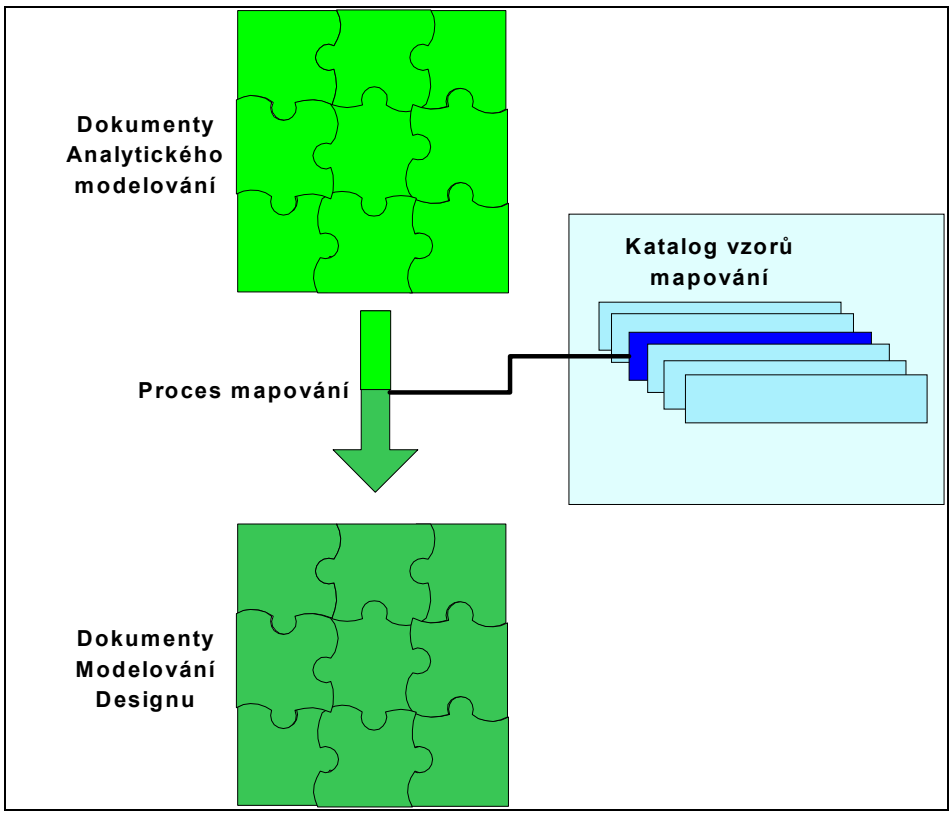
Díky těmto dvěma bodům lze velmi rychle přecházet z úrovně analytického modelování AM do designu D.

První bod zabezpečuje jednoznačnost, jednotnost a přesnost dokumentace analytického modelování AM, čímž vzniká přesné a jednoznačné zadání pro design.

Druhý bod zefektivňuje přechod z AM do D opakujícími se postupy mapování, takže mnohdy dokumentace designu bývá tvořena pouhým odkazem na zvolený postup ze seznamu vzorů.

Co se týče dalšího přechodu z úrovně abstrakce modelování designu do kódování (tj. sama realizace z D do C), jeho efektivita je dána efektivitou použití nástrojů daného konkrétního vývojového prostředí.

Postup vedoucí k efektivitě a rychlosti tvorby informačního systému při přechodu od AM do D ukazuje následující obrázek:



obrázek 4 Postup mapování je dán katalogem vzorů

Pracovník tvořící modely designu D (tmavě zelená oblast) vychází jednak z modelů analytického modelování AM (světle zelená oblast) a současně vybere a použije již zdokumentovaný vzor mapování do designu odkazem do seznamu všech vzorů mapování (modrá oblast). Dokumentace se tak skládá ze tří částí:

- dokumenty analytického modelování (AM)
- dokumenty designu (D)
- použitý výskyt vzoru mapování z AM do D

Tohoto přístupu se použije vždy a bez výjimky, a to i kdyby se daný vzor mapování použil pouze jednou. I když na něj bude pouze jeden odkaz, tak i v tomto případě je třeba tento jinak unikátní vzor umístit do katalogu a odkázat se na něj.

Seznam možných vzorů mapování se tak v životě firmy rozšiřuje a firma tak stále více efektivněji přechází z AM do D.

1.19 Analytické modelování jako abstraktní programování

Tvorba na úrovni abstrakce AM je velmi obtížná a vyžaduje zkušené pracovníky - analytiky. Tvorba dokumentů AM podléhá svým přesným pravidlům. Syntaxe AM umožňuje velmi přesně a velmi podrobně popsat systém, aniž by se přitom používaly prvky z dané konkrétní (vývojářské) technologie, tj. prvky z designu. K vyjadřování se používají abstraktnější pojmy v notaci UML. Celý systém se takto popisuje ve své úplnosti na abstraktnější úrovni. Výsledkem je abstraktní obraz informačního systému, který se poté realizuje v dalších nižších úrovních abstrakce, tj. „programuje se na základě dobrého zadání“.

Tvorbu modelů AM lze přirovnat k ***abstraktnímu programování***. Výsledkem je „implementačně nezávislý abstraktní program informačního systému“. Modely z AM lze proto považovat za ekvivalent abstraktního obrazu budoucího kódu. Pokud se používá pojem „model informačního systému na úrovni AM“, má tím autor na mysli „abstraktní model konkrétního napsaného programu“.

1.20 Modely podniku BM (BUSINESS MODELING) a jejich vztah k analytickému modelování AM

Pro úroveň abstrakce AM se někdy používá také zkrácený název „analýza“. Role pracovníka, který tvoří modely z této úrovně, se nazývá role analytika. Protože se mnohdy pojem analýza používá v jiných souvislostech a v jiném významu v různých metodikách, dochází tak často ke kolizi pojmů. Tomuto problému je věnována tato kapitola.

Někdy se pod pojmem analýza skrývá tzv. modelování podniku, tj. BUSINESS MODELING, dále také zkratka BM. I v tomto případě se jedná o modelování, ale liší se od analytického modelování AM zásadně a to předmětem modelu, tj. liší se tím, co se vlastně modeluje.

Zatímco v analytickém modelování AM je předmětem modelu zkoumaný informační systém, tj. jedná se vlastně o abstraktní obraz programu (viz *obrázek 1 Úrovně abstrakce informačního systému*), u BM je předmětem samo prostředí, tj. podnik, do kterého má být informační systém dosazen. Protože dané prostředí podniku je také objektově orientované, lze pro toto modelování použít také jazyk UML. Objekty v modelech BM jsou zde objekty daného prostředí (pracovník, vedoucí, oddělení, oběžník, obsluha, knihovník apod.). Z tohoto modelování má významné postavení tzv. BUSINESS PROCESS MODELING, zkráceně BPM. Nosným prvkem tohoto modelu jsou aktivity podniku neboli procesy podniku.

Modelování podniku se provádí dost často a to hlavně z těchto důvodů:

- Je třeba vyvinout informační systém a nejsou dobře známy procesy podniku, které tento informační systém bude podporovat. V tomto případě se jedná o pohled na modely podniku z hlediska softwarových vývojářů, konkrétně analytiků, kteří „pátrají“ po funkcionalitách budoucího informačního systému. Modely podniku napomáhají zprostředkovaně nalézt tyto funkcionality systému. Specifické postavení má toto modelování jako zvláštní technika pro vyhledání všech případů užití (prvky USE CASE) informačního systému (dále bude pojednáno v odpovídající kapitole tvorby dokumentu USE CASE MODEL).
- Je třeba navrhnout nové procesy podniku (například nová služba banky, nová služba operátora mobilních telefonů apod.) a současně navrhnout nové

funkcionality informačního sytému. V tomto případě se problém také týká softwarových pracovníků.

- Je třeba optimalizovat procesy podniku, například navrhnout nový způsob koordinace přepravy, optimalizace chodu podniku, chodu skladu apod. Artefakty této práce jsou přehledné modely v grafické podobě, které srozumitelně a jasně popisují chod jak původních, tak optimalizovaných procesů podniku.

Pro efektivní tvorbu IS jsou důležité body 1 a 2, zatímco bod 3 spadá pod kompetenci konzultačních firem optimalizujících chod podniků.

Pokud dodavatel softwaru nezná procesy podniku, tj. jak to má v daném prostředí správně chodit, nemůže navrhnout dobrý informační systém. Z toho důvodu některé softwarové firmy dodávají modelování podniku BM jako součást vývojových prací na informačním systému. Na základě praxe v několika desítkách firem je třeba upozornit na některá úskalí tohoto modelování.

1.21 Chyba samostatných prací na modelování podniku bez ohledu na funkcionality budoucího IS

V prvé řadě je třeba upozornit na skutečnost, že nelze vytvořit pouze model podniku BM a nezohledňovat přitom navrhovaný informační systém. V některých firmách byl v projektu zvolen tento postup: Úvodní část projektu (asi okolo 10%) se věnovalo modelování podniku a poté se přistoupí k pracím na vývoji SW (AM, D, K). Rád bych upozornil na nebezpečí spojené s tímto postupem.

Je nezbytné tvořit oba modely, jak business model BM, tak analytický model AM souběžně a neustále přecházet od modelu podniku BM k modelům AM a zpět. V opačném případě hrozí, že bude vykonána spousta zbytečné práce a projekt tvorby informačního systému bude ohrožen.

V případě, že firma bude nejprve modelovat podnik a poté informační systém, hrozí tato rizika:

- Existuje zřejmá a silná zpětná vazba mezi navrhovaným modelem informačního systému a modelem podniku. Podle toho, jaké se navrhne řešení informačního systému, může se zpětně návrhem IS ovlivnit chod podniku. To je nejvážnější riziko postupu modelování podniku BM, které nezohlední návrh IS.

- Samotné modelování podniku vyvíjené bez ohledu na funkcionalitu informačního systému nemusí mít pevně stanovené hranice. Pod jeho působnost může spadat spousta jinak nezajímavých věcí, například lze do modelů podniku zahrnout nejenom již dále neřešené oblasti, ale je možné také zbytečně „rozpitvávat“ nezajímavé detaily. Nejenže to stojí čas a peníze, ale navíc zbytečná práce má nepříjemný dopad na tvůrčí atmosféru v týmu.
- Mnohdy nastává nepříjemná situace, že model podniku BM je zaměněn s modelem informačního systému AM resp. naopak. Jeden pracovník hovoří o prvcích z podniku a druhý pracovník chápe tyto prvky jako prvky informačního systému resp. obráceně. Obzvlášť nepříjemná je tato záměna při vztahu mezi dodavatelem a odběratelem softwarového řešení, kdy dodavatel odevzdává model podniku a odběratel jej chápe jako model informačního systému. To může vést k vážným kolizím při dohodě nad funkcionalitami systému. Je třeba upozornit na to, že tyto záměny bývají velmi záludné. Dokonce se jedná o jednu z nejčastějších chyb analytika. Důvod je prostý: Pojmy jak v modelování podniku BM, tak v analytickém modelování AM znějí stejně, ale každý z nich má jiný význam. Chyby jsou takto dobře maskovány shodou názvů pojmů v obou oblastech modelování. Například pojem směnka v bance a evidovaná směnka v IS mají stejný název, ale jsou to dva oddělené pojmy ze dvou různých modelů, řečeno fyzikálně ze dvou „prostorů“. Jeden prostor je prostředí podniku a druhý prostor je vnitřek informačního systému. Záměny tohoto typu jsou velmi snadno přehlédnutelné. Praxe ukazuje, že uvedená chyba je charakteristická zejména pro modelování stavových modelů u evidenčních systémů. Uvedené problematice záměny AM versus BM je věnován jednak příklad v této kapitole a také příklad v kapitole o stavových modelech.

Některé firmy zahajují vcelku logicky práce na modelech podniku BM a poté přistupují k samotnému řešení problematiky IS, tj. k tvorbě modelů AM. Jak bylo řečeno, mnohdy bývá určitá úvodní část projektu vyhrazena analytickým pracím v oblasti modelování podniku BM. Projekt tvorby IS se takto naplánuje podle schématu: Prvních X týdnů budeme tvořit modely BM a poté, po vyhotovení modelů BM, přistoupíme k řešení IS. Osobně ze zkušenosti z konzultací z firem nedoporučuji až takto oddělit práce na modelování podniku BM a analytickém modelování AM. Z důvodu efektivního modelování doporučuji, aby se modely BM tvořily souběžně s modely AM a nikoliv nějak extra dopředu jako zvláštní úvodní část projektu z uvedených důvodů rizik, které tímto nastávají. Zejména problém zpětné vazby může vést k velmi nečekaným komplikacím.

Model podniku BM je sice mnohdy velmi důležitý a pro projekt dokonce velmi často i nezbytný, ale z hlediska vývoje SW je třeba jej chápat pouze v rovině sekundární informace, tj. „proč je model informačního systému takový, jaký je“. Cílem efektivní technologie tvorby SW je získat dobrý model AM. Designér se opírá pouze o artefakty AM samotného informačního systému a z nich dostává úplné a postačující

informace pro svou tvorbu. Designéra modely podniku BM vůbec nezajímají, protože jsou mimo rozsah jeho řešení. Designér považuje BM za „zbytečnou omáčku“. Častou chybou analytika bývá dodání dokumentace modelů BM a prohlášení těchto modelů za modely AM. Designér tak dostává spoustu zbytečných informací o samotném podniku (nikoliv o IS), ze kterých lze teprve dedukcí vyčíst, co má vlastně program dělat. Designér tak musí tyto informace přebrat, vyhodnotit a vytvořit z nich pro sebe analytický model programu, což měl původně dostat jako zadání od analytika. Teprve poté přistupuje k designu, přičemž dokumenty analytického modelování vůbec nevznikly.

Je třeba zdůraznit, že modely podniku BM mají své důležité místo v jiných částech projektu, například při popisu chodu služby podporované informačním systém, což si odběratel systému určitě rád přečte a posoudí správnost modelu podniku.

1.22 Příklad na zpětnou vazbu modelování podniku a informačního systému

Představme si následující situaci. Analytik, který navrhuje systém, se baví s budoucím uživatelem o požadovaných funkcionalitách informačního systému. V rozhovoru zazní od uživatele tato věta:

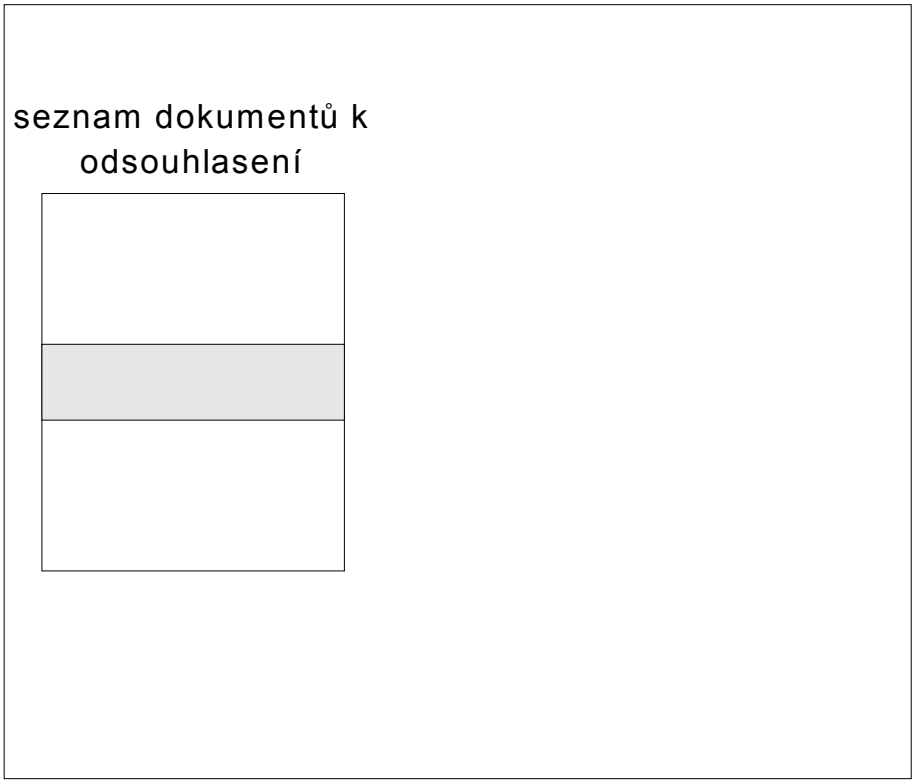
„...poté se daný dokument odsouhlasí odpovědným pracovníkem a tento se odešle z podniku... (samozřejmě dokument, nikoliv pracovník)“

První věc, kterou musí mít analytik vyřešit, je odpověď na otázku: Kam spadá tato věta? Je to věta z modelování podniku BM (tj. hovoříme o chování podniku), anebo je to věta z modelování informačního systému AM (tj. hovoříme o evidenci v IS)? Odpověď v tomto případě zní: Mluvíme o chování podniku, tj. jedná se o větu z oblasti modelování podniku BM (dokonce přímo z oblasti BUSINESS PROCESS MODELING). Zatím se nebavíme o chování samotného informačního systému, ale o chování podniku.

Analytik hovoří s budoucím zákazníkem a navrhuje: Nabízejí se tato řešení (a nyní se již hovoří o informačním systému).

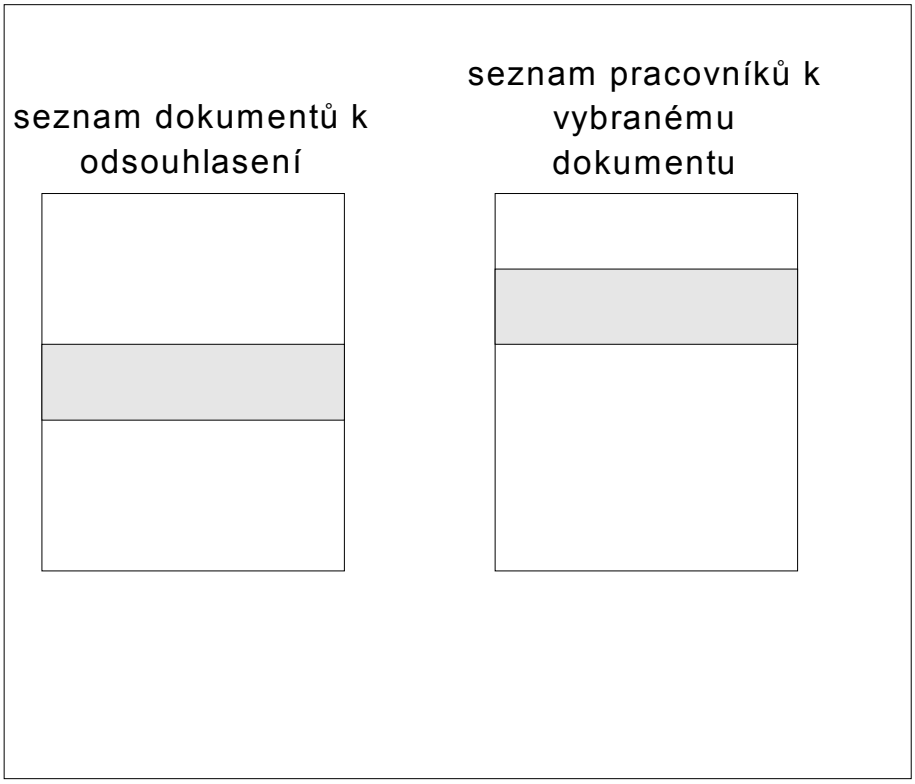
„Představme si, že systém bude fungovat takto: Obsluze se zobrazí seznam dokumentů k odsouhlasení, obsluha vybere dokument.“

Analytik namaluje náčrtek, jak si analyticky představuje obsah obrazovky, např. takto:



obrázek 5 První náčrtek analytika při návrhu funkcionality

Analytik pokračuje: *„Dále se obsluze nabídne seznam odpovědných pracovníků, kteří mohou daný dokument podepsat a obsluha vybere pracovníka.“* Domaluje další část náčrtku obrazovky:



obrázek 6 Pokračování náčrtku analytika

„Po vybrání obsluha potvrdí (například OK tlačítko apod.), že tento vybraný pracovník odsouhlasil tento vybraný dokument a tato informace „tento pracovník odsouhlasil tento dokument“ se uloží (možná ještě s dalšími informacemi, jako kdy to odsouhlasil, kdy to bylo zavedeno do systému, kdo to tam zavedl apod.)…To je však pouze první možná varianta řešení“, pokračuje analytik.

„Existují další varianty, které bychom vám mohli dodat. Druhá varianta se liší od předešlé tím, že v systému je zavedena agenda přístupových práv. K odsouhlasení dokumentu dojde takto: Daný pracovník, který má odsouhlasit dokument, se musí přihlásit do systému jako uživatel. Přes agendu přístupových práv se díky tomu, kdo je přihlášen, přihlášené obsluze, tj. pracovníkovi, zobrazí ty dokumenty, které může jako pracovník - přihlášený uživatel odsouhlasit. Obsluha (což je nyní daný zodpovědný přihlášený pracovník) označí dokument k odsouhlasení a potvrdí.

Třetí varianta dále rozvíjí předešlou: Agenda přístupových práv je tvořena přes technologii asymetrického podpisu s privátním a veřejným klíčem. Znamená to, že přihlášení probíhá s podporou této technologie a informace tento pracovník odsouhlasil tento dokument je navíc vybavena podpisem pomocí privátního klíče daného pracovníka.“

Třetí varianta se od předešlé liší pouze „modernější“ a zatím neprolomitelnou technologií zabezpečení asymetrického podpisu.

Všimněme si na tomto příkladu „silné“ zpětné vazby, spočívající v tom, že podle toho, jaké řešení zvolíme, tak se zpětně ovlivní chod podniku a následně se najdou nové funkcionality systému.

Zpětná vazba se projeví v tom, že pokud se přijme řešení druhé, resp. třetí, přibývají další nové procesy podniku a následně nové funkcionality systému, které jsou spojeny s podporou tohoto řešení. Stejně tak chování firmy se výrazně změní. Například otázka: „Daný pracovník odjel na dovolenou“ (věta z BM) se v systému projeví mnohem vážněji v druhém a třetím případě. V prvním případě stačí, když podepíše daný reálný papírový dokument a odjede, ve druhém a třetím musí on sám provést záznam do systému. Pokud se přijme druhé řešení, systém musí mít celou agendu přístupových práv a s tím spojené procesy podniku, ve třetím případě ještě navíc správu klíčů apod. Procesy v podniku a následně další funkcionality systému musí tyto skutečnosti zohlednit. Jinak řečeno návrhem systému se ovlivňují další jiné oblasti řešení v jiných oblastech BM a následně také IS.

Proto zásadně nedoporučuji, aby se nejprve provedla celá analýza podniku a poté se přistoupilo k řešení informačního systému. Nasazení systému totiž ve svém zvoleném řešení ovlivní zpětně chování podniku a následně vyvolá nutnost posouzení dalších funkcionalit systému.

Dalším názorným příkladem zpětné vazby v návrhu IS a BM je nalezení různých automatizovaných procesů, které systém podporuje a které zpětně ovlivní chování podniku. Určitě bude chod úvěrů v bance vypadat jinak bez použití informačního systému, než s použitím informačního systému. Jinak řečeno chod úvěrů bude zpětně záviset na přijatém řešení IS. Nezohlednit tuto jednoduchou myšlenku, tj. zvážit, co může systém nabídnout díky automatizaci a zpracování, by vedlo k fatálním chybám v modelech.

Dalším příkladem silné zpětné vazby je návrh systémů, které využívají moderní technologie komunikací jako jsou web, mobilní telefony apod. Pokud navrhujeme například službu mobilního operátora (například proplacení lístků do kina při objednání přes web a platbě mobilním telefonem), nelze tuto službu dobře navrhnout bez současného návrhu funkcionalit IS.

1.23 Příklad na záměnu analytického modelování a modelování podniku

Další častou chybou je záměna analytického modelování a modelování podniku. Jako příklad se představme situaci, kdy analytik hovoří s budoucím uživatelem a dozví se následující informaci: „Faktura k proplacení, která přijde do podniku, musí mít alespoň jeden řádek“.

Opět si musíme položit otázku: Kam spadá tato věta, do kterého prostoru? Do modelování podniku BM nebo do analytického modelování informačního sytému AM? Je zřejmé, že se jedná o větu z prostoru modelování podniku BM, protože se hovoří o faktuře podniku a nikoliv o evidované faktuře.

Zkusme tuto větu, která je z modelování podniku, „otrocky“ převzít do návrhu systému. Zavedeme také u evidované došlé faktury omezení, že vyplněnou fakturu nelze do systému „přijmout“, dokud nemá alespoň jeden řádek.

Představme si, že obsluha vyplní celou hlavičku faktury, což trvá například poměrně dlouho, podívá se na hodinky a řekne: „OK, zítra dodělám řádky a zvolí *Uložit*“. Systém ji však nesmyslně nepustí dál a požaduje vyplnění alespoň jednoho řádku.

Řešení je samozřejmě triviální: Zavedeme stavy evidované došlé faktury, jedním z těchto stavů bude například „otevřena k editaci“ a dalším „editace uzavřena“. Ve stavu „otevřena k editaci“ může obsluha provádět libovolné úpravy faktury. V okamžiku, kdy zvolí „uzavírání faktury“, tak v tomto okamžiku, tj. v tomto přechodu, by se mělo kontrolovat, zda existuje alespoň jeden řádek. Informace „došlá faktura má alespoň jeden řádek“ tedy neznamená omezení ve smyslu, že evidovaná faktura musí mít také alespoň jeden řádek, ale vede k logice „jak se má evidovaná faktura chovat, aby se tento požadavek splnil i v evidenci“.

I když se jedná o triviální příklad, tato chyba „otrockého“ převzetí podmínek z modelování podniku BM do analytického modelování AM je, jak jsem zjistil z konzultací ve firmách, velmi častá. Příčina chyby je v tom, že entita z okolí, v tomto příkladu „došlá faktura do podniku“, a entita ze systému, zde „evidovaná došlá faktura“, spolu sice logicky souvisejí, ale v žádném případě nejsou totožné (fyzikálně řečeno „každá entita patří do jiného prostoru“). Určitě nejsou v nějakém vztahu „podmnožiny“. Například entity evidované v systému mají stavy, které odpovídají tomu, že žijí v prostoru IS, přičemž tyto stavy v podnikových entitách nenalezneme. Stav „editace uzavřena“ je stavem evidované faktury v systému (nikoliv okolní entity v podniku). Tento stav je pro fakturu v podniku irelevantní. Takovýchto rozdílů

bychom mohli najít pochopitelně více, jsou dány rozdílnými vlastnostmi prostorů podniku a informačního systému.

Pro zamezení této chyby doporučuji, aby analytik buď nahlas anebo pouze v duchu u každého pojmu, o kterém je řeč, vždy dodal: „evidovaná“ anebo „z podniku“ a tím vždy rozlišil, ve kterém prostoru se pohybujeme. V předešlém příkladu by tedy úvodní věta, která původně mohla vést ke zmatením pojmů, měla znít:

„Došlá faktura v modelu podniku musí mít alespoň jeden řádek“. Zdůraznili jsme „v modelu podniku“, následuje proto vcelku jasná a logická otázka: Jak se tato vlastnost projeví u „evidované došlé faktury v systému“? Odpověď zní: Zavedeme u evidované faktury stav „otevřena k editaci“ a možnost přechodu do stavu „editace uzavřena“. V tom okamžiku se bude kontrolovat, zda má evidovaná faktura alespoň jeden řádek. Samozřejmě tento mechanismus nevylučuje možnost přechodu zpět do stavu „otevřena k editaci“ pro opravy.

1.24 Analýza jako zdroj informací

V některých případech se pod pojmem analýza neskrývá pojem analytické modelování, ale tvorba souhrnu dokumentů, které nějak pojednávají o dané problémové doméně a mají k řešení nějaký relevantní vztah. V tomto případě má pojem analýza význam „tvorba zdroje informací“. V žádném případě se nejedná o tvorbu dokumentace AM.

1.25 Příklad na analýzu jako zdroj informací

Když jsem nastoupil do zmíněné firmy vyrábějící bankovní software, jako první úkol jsem dostal vypracovat analýzu šekové agendy. V té době jsem pochopitelně neměl ani ponětí o tom, co je to „analýza“, natož analytické modelování.

Posbíral jsem tedy všechny tehdy platné zákony, předpisy a dokumenty, které se týkaly šeků. Sepsal jsem poměrně rozsáhlý dokument s názvem „Analýza šekové agendy“. Uvnitř dokumentu bylo vcelku podrobně pojednáno, jak se pracuje s šeky, jaké jsou typy šeků (zaručené, cestovní apod.), co je ze zákona povoleno a co zakázáno apod.

Tento dokument jsem odevzdal svému šéfovi, který byl s tímto dokumentem dokonce velice spokojen se slovy „konečně někdo napsal nějakou analýzu nějaké agendy!“.

Avšak první programátor, který mne potkal na chodbě, mne vrátil nohama na zem: „A co si myslíš, že podle tvého dokumentu mohu naprogramovat?“

Analytické modelování AM je o něčem jiném, než uvedený dokument o šekové agendě: V AM se jedná o model, který skutečně slouží jako velmi přesné a úplné zadání pro design, tj. pro technology a programátory, protože popisuje samotný program (viz pojem *analytické programování*).

Oproti tomu analýza jako zdroj informací je pouze souhrnem dokumentů, který slouží k dobré orientaci v dané problémové doméně. Mezi tyto dokumenty analýzy jako zdroje informací patří například souhrn všech zákonů a předpisů z dané oblasti, výsledky konzultací se zákazníkem, uspořádané požadavky na systém apod. Na rozdíl od toho analytický model z AM pojednává již konkrétně o samotném programu, tj. jaké informace systém konkrétně eviduje, jak se chovají výskyty těchto informací, co dělá obsluha se systémem apod.

Netvrdím, že není mnohdy velmi prospěšné analýzu jako zdroj informací provádět, například určitě je vhodné provádět správu požadavků. Jedná se však o výchozí materiály pro tvorbu AM. Designéra však vždy zajímají až výstupy z analytického modelování AM jako analytický návrh systému, to chápe jako jeho zadání.

1.26 Podcenění analytického modelování a časované bomby v projektech

Mnoho firem podceňuje fázi tvorby analytického modelování AM. Mnohdy je tato fáze zdokumentována jen slabě anebo dokonce vůbec ne. Na první pohled se zdá, že práce na analytickém modelování mohou zpomalit tvorbu softwaru, ale praxe ukazuje, že opak je pravdou.

V prvé řadě je třeba si připomenout skutečnost, že každý vyvíjený software vždy prochází fází tvorby v úrovni abstrakce analytického modelování. Nelze si představit situaci, že by byl nějaký software napsán bez toho, že by se autor vůbec nezamyslel nad tím „Co vlastně programuje“. Přitom ona otázka „CO“ je výstižnou zkratkou pro práce v oblasti analytického modelování. Takže výsledky této fáze existují vždy a je jenom otázkou, zda tyto výsledky zůstanou v hlavách autorů anebo se pokud možno efektivně a rychle zdokumentují. Chybějící dokumentace analytického modelování je nutně nahrazována sekundární tvorbou „ústních vysvětlení“ tvořených přímo na místě. Stabilní artefakt dokumentace je mnohdy chybně nahrazen nestabilními a dočasnými artefakty rozhovorů se všemi z toho plynoucími zápornými důsledky.
V tom lze také spatřovat nejnepříjemnější efekt chybějící dokumentace analytického

modelování: Artefakty vývoje nejsou samostatně stojící, nelze je považovat za jeden zapouzdřený „balík“.

Důsledky neúplné dokumentace, v tomto případě chybějící dokumentace analytického modelování, jsou nasnadě. V podstatě vedou k již uvedeným efektům metody řízení projektů TUNEL. Projekt se stává nepřehledným a je patrná nízká transparence výsledků. Vznikají nelogičnosti, chyby, nepřehlednosti, tj. artefakty vývoje jsou nečitelné resp. čitelné velmi obtížně. Pracovníci se stávají nezastupitelnými, musí být neustále po ruce, nelze je odvolat na nové projekty, i když „starý“ projekt je již hotov. Velmi pracně se zavádějí opakovatelné postupy. Je znatelný vysoký stupeň chaosu ve vývoji. Vznikají softwarové slepence a nedodělky. V důsledku těchto jevů rostou stále větší nároky na operativní řízení. Díky nízké transparenci, nečitelnosti, díky opakování prací a tvorbě zbytečné práce se zvyšuje hektičnost prací, včetně nároků na volný čas apod. Jevy zde popsané se stávají časovanou bombou a mají pochopitelně negativní vliv na celou firmu.

Mnohdy se k programování přistupuje se slovy: „Analytické modelování je příliš složité, nutí nás odhalovat všechny detaily systému na logické úrovni, a ty ještě neznáme, a vůbec je toho moc… Zdržuje to! Pišme raději program!“ Taková situace, i když je vlastně nelogická, bývá ve firmách běžná. Neví se, jak má daný systém fungovat, o systému panuje všeobecná neznalost, kterou se nedaří odstraňovat a práce proto stojí. Tak už konečně začněte programovat!

Tady je třeba zdůraznit jednu důležitou věc: Pokud panuje o systému všeobecná neznalost, tak je to pro daný projekt skutečně „červená kritická situace“, která je srovnatelná s alarmem v nebezpečných provozech. Tato situace by měla vést ke zvýšenému úsilí a nasazení, avšak nikoliv programátorů, ale analytiků. Musí se velmi rychle provést analýza stavu projektu, v čem je příčina zbrždění prací a co se v projektu děje. Příčin může být několik: neschopný analytik, špatný konzultant, špatně dohodnuté podmínky s odběratelem, který může ze dne na den cokoliv „z gruntu“ změnit a odvolat včerejší stoprocentní pravdy apod. V každém případě tento kritický stav projektu není důvodem ke zvýšení úsilí programátorů, ale analytiků případně vedoucích.

1.27 Analytické modelování a přechod na nové technologie

Existuje ještě jedna velmi často se vyskytující situace, která očividně ukazuje na nutnost analytického modelování. Stává se, že firma potřebuje přejít z jedné starší technologie na technologii novou, například ze zastaralé dvojvrstvové na třívrstvovou s využitím WEB serveru při takřka stejných požadavcích na analytické úrovni. Vzniká problém: Jak tento přechod na novou technologii uskutečnit?

Ukazuje se, že nelze jen tak přejít z jedné technologie na druhou pouze změnou designu. Přejít „z jednoho technologického návrhu přímo do druhého“ by znamenalo napsat nějaký „automat – konverter“, který by vzal jeden kód a zkonvertoval by jej do druhého kódu (z jednoho D do druhého D), podobně jako kompilátor tvoří ze zdroje spustitelné soubory. Například původní kód ve FOXPRO by byl automatem zkonvertován do kódu C# (nebo naopak) apod. Protože takto postupovat nelze, je zřejmé, že pracovníci při přechodu musejí znát „co vlastně programují“ a tedy musejí znát výsledky analytického modelování. Jednoduše řečeno musí si říct „o co tam vlastně jde“, což je podstata AM.

Pokud existují zdokumentované artefakty AM, pak by se v podstatě jednalo o jednoduchý projekt nového mapování ze stejného analytického modelu AM do jiného nového designu D. Pokud však dokumentace analytického modelování AM neexistuje (tj. AM existuje pouze v hlavách tvůrců), tvůrci designu modernější technologie se dostávají do již známé situace: Jsou nuceni předávat si své poznatky z úrovně analytického modelování pouze ústně a opět se zde při přechodu na nové technologie objevuje celá škála problémů metody TUNEL.

1.28 Role v projektu: analytik, designér a programátor

Rozdělení prací do třech úrovní abstrakce s sebou přináší také rozdělení rolí ve vývojovém týmu.

Podle těchto úrovní abstrakce lze rozlišit tyto tři role:

- analytik (tvoří modely analytického modelování, příp. také BPM)
- designér (tvoří modely designu, mapuje z analýzy do designu)
- programátor (tvoří samotný chodící kód)

Následuje seznam nutných znalostí, které by pracovníci v těchto rolích měli ovládat. Pomocí tohoto výčtu dovedností lze také lépe pochopit význam odpovídajících abstraktních úrovní.

1.29 Analytik

Analytik potřebuje zejména zvýšenou až dokonalou znalost UML, tj. oproti ostatním rolím je guru na UML, zná syntaxi UML do takřka všech detailů včetně příslušného CASE nástroje. UML je jeho hlavní „abstraktní programovací jazyk“. Současně se znalostí UML by měl mít velmi dobře rozvinuté abstraktní a kreativní myšlení s vysokou představivostí. Analytik je schopen „vidět budoucí systém před sebou“. K tomu, aby byl schopen rychle předat ostatním členům týmu tuto představu, musí umět dobře a rychle modelovat, tj. musí mít schopnost rychle a přesně formulovat myšlenky a zapsat je do modelů.

Je vcelku pochopitelné, že analytik má dobré znalosti z problémové domény, kam informační systém spadá. Pokud existuje odběratel, analytik spolupracuje s konzultantem formou tzv. interview neboli pohovorů. Díky těmto pohovorům vznikají požadavky na systém. Přitom nelze opominut ještě jednu důležitou vlastnost, která vyplývá z toho, že analytik je v kontaktu se zákazníkem při určování funkcionalit systému, kdy pomocí takovýchto „interview“ určuje budoucí chování systému. Analytik musí mít také odpovídající schopnost komunikace s uživatelem. Jsou na něj kladeny zvýšené nároky ohledně schopnosti asertivního jednání se zákazníkem. Velmi schopný analytik je schopen „dotlačit“ uživatele ke správnému řešení a to dokonce bez konfliktů a kolizí. Podotknu pouze, že mnohdy je to nadlidský úkol, zejména pokud se na straně odběratele setkáme s člověkem jak neznalým, tak neústupným.

Poznámka: Mnoho programátorů nemá schopnost asertivního jednání, dokonce většina se raději takovýmto jednáním snaží vyhnout. Jedním z důvodu je, že většina programátorů jsou povahou spíše introverti, tj. jsou málo společenští a raději pobývají v tichu pracovny a píší programy. Jiným důvodem je často se vyskytující určitý druh arogance v obci programátorské. Dlouho jsem bádal nad tím, čím to je, že v programátorské branži se vyskytuje tak vysoké procento jedinců, kteří jsou výrazně nesmlouvaví k chybám resp. neznalostem druhých (viz některé diskuse na internetu). Myslím, že jedním z důvodů je to, že programátoři jsou dlouho vychováváni kompilátorem, který jim neodpustí ani čárku. Chybou je, když tuto vlastnost „velmi arogantní nesmlouvavosti kompilátoru“ přenášejí také do mezilidských vztahů s určitým nadřazeným pohledem vůči ostatním. Zažil jsem například osobně tuto situaci: Sedíme na jednání se zástupci banky - budoucího zákazníka a dostáváme do ruky jejich dokument. Jeden z mých kolegů programátorů ohodnotil dokument předaný ze strany banky slovy: „Toto mohl napsat jenom debil...“ Přitom autor toho dokumentu seděl na jednání. Mimo místnost jsem dotyčnému sdělil, kdo je vlastně debil a proč.

1.30 Designér

Designér se na systém již dívá trochu jinýma očima. Pro něj je tvorba systému již technologickým úkolem: Má zadání od analytika, umí a zná dané vývojové prostředí, takže vyvstává otázka designéra: „Jak tuto analýzu, tento analytický model AM, obsahující vše co potřebuji, navrhnu v mém známém prostředí“? Pro něj jako experta na technologii softwaru je sama podstata problému v analýze vlastně nezajímavá. Slouží mu pouze jako zdroj (zadání) pro jeho tvorbu v oblasti technologické. Jako dobrý technolog je schopen navrhnout jakýkoliv systém, když má dobré analytické zadání. Oním dobrým zadáním jsou právě výsledky práce analytika. Synonymem pro roli designéra je někdy používaný název role „technolog“ anebo „architekt“. Designér je hlavní guru na danou vývojovou technologii. V některých případech je problematika vyvíjeného softwaru technologicky natolik složitá, že se na projektu musí podílet hned několik designérů. Každý z nich je specialistou na určitou oblast technologie (např. databáze, bezpečnost, operační systémy, WEB apod.). Znalosti designérů vystihuje tento seznam:

- znalost UML jako dobrý pasivní čtenář
- databázová teorie, návrh ERD resp. jiné databáze (stromová apod.)
- konkrétní vlastnosti používané databáze (administrace apod.)
- principy návrhu GUI daného prostředí
- problematika bezpečnosti daného prostředí
- operační systém
- sítě
- HW pro návrh nasazení SW
- Web

... a jiné speciální znalosti z technologií SW.

V neposlední řadě je třeba zmínit také znalost již známých vzorů mapování, tj. schopnost použít již jednou použitých postupů mapování z analytického modelování do designu.

1.31 Programátor

Pro programátora je tvorba systému již „pouhé“ napsání a rozchození daného kódu. Z toho důvodu musí mít tyto dovednosti:

- znalost UML jako dobrý pasivní čtenář
- velmi dobrá znalost vývojového prostředí, dobrá orientace a znalost všech možností tohoto prostředí atd.
- velmi dobrá znalost syntaxe daného jazyka (např. Java, C#, apod.) a schopnost rychle psát a samostatně po sobě testovat kód v daném prostředí
- znalosti syntaxe spolupráce s danou databází (např. SQL syntaxe apod.)
- základní znalosti designéra (nikoliv detailní)
- schopnost psát rychle dobrý a funkční kód

Povaha prací na všech třech úrovních se podstatně liší: Zatímco analytik je spíše „spisovatelem“, který se pohybuje na vyšší úrovni abstrakce, designér jako technolog je typický představitel technokrata, který si libuje ve finesách daného prostředí. Programátor zde vystupuje již jako kvalitní a spolehlivý realizátor programu.

Je vhodné, aby se v různých projektech programátoři účastnili fáze designu, například veřejnou oponenturou resp. diskusemi, rotací v této roli apod. Práce programátora se totiž může stát v tomto rozdělení prací nezajímavá, protože není příliš kreativní a připomíná spíše „vyšívání kódu na dílně“.

Rozdílná povaha prací mnohdy způsobuje velmi velké problémy u těch vývojářů, kteří musí pracovat v několika rolích současně. Jedná se o ten nešťastný případ metody příčného řezu, kdy pracovník je povinen odevzdávat výsledky podle pokynu „sám analyzuj, sám navrhni design v dané technologii a sám naprogramuj“. Takovéto pendlování mezi rolemi vyžaduje, aby se daný pracovník v určité chvíli choval jako analytik, poté se změnil na designéra a poté na programátora. Přitom však povahy těchto rolí jsou diametrálně odlišné. Nejenom, že se tímto pendlováním zabraňuje specializaci, kdy každý umí všechno (například firma má experta na relační databázi ORACLE, který současně velmi dobře ovládá problematiku zahraničních akreditivů), ale navíc rozdílná povaha prací vede i k vnitřním psychologickým konfliktům. Každý pracovník většinou tíhne k určitému typu práce, například více analytické a méně technologické anebo naopak. Pendlování mezi těmito rolemi může způsobit určité kolize při práci v jiné roli, než v té, která danému pracovníkovi lépe vyhovuje.

Klasický technolog se raději vyhýbá fázi analytického modelování, klasický analytik se zase velmi nerad řeší podrobnosti dané technologie. Při odhalení funkcionality by se rád zabýval dalšími otázkami nových funkcionalit.

Představme si osobu, která je na jedné straně vynikající expert na databáze, zná velmi dobře například ORACLE nebo MS SQL, a to včetně všech „moderních fines“. Na straně druhé je současně velmi dobrý znalec na účetnictví. Zná všechny „byrokraticky hnusné“ vyhlášky z účetnictví a také všechny jejich výjimky. Pro představu například ví velmi dobře, jak se vystaví faktura o došlé platbě za zálohovou platbu s DPH, když odběratelem je firma ze Slovenska jako plátce DPH, přičemž služba byla poskytnuta na území Německa provozovnou společnosti se sídlem v ČR. Takového člověka, který umí jak administrovat databázi, tak současně vyřeší předešlý dotaz z účetnictví, bych považoval spíše za schizofrenika, než za „normálního člověka“, protože tyto dvě osoby se v něm musí zákonitě pobít. Ono vlastně upřímně řečeno, s takovýmto „rozpolceným“ člověkem jsem se ještě nesetkal ☺.

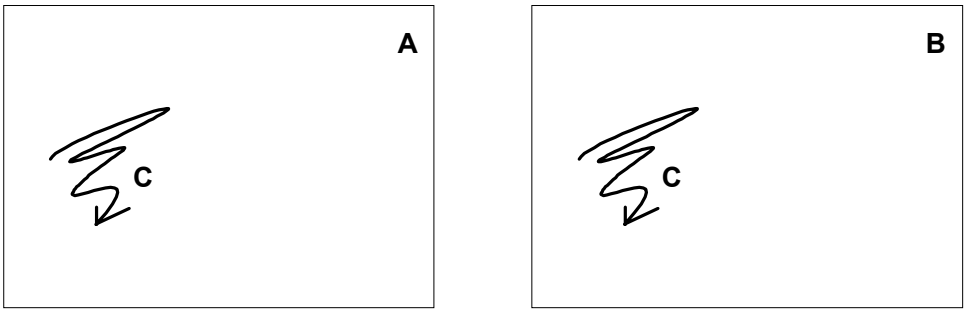
2. Objektově orientovaný přístup (OOAP)

V této kapitole jsou vysvětleny základní principy objektově orientovaného přístupu. Bez znalosti těchto principů není možné zavést kvalitní tvorbu SW.

2.1 Princip opětovné použitelnosti (re-use)

Jedná se o jednoduchý základní axiom, jehož důsledky se linou teorií tvorby SW. Pod opětovnou použitelností, kterou budeme nazývat také re-use, máme na mysli následující situaci:

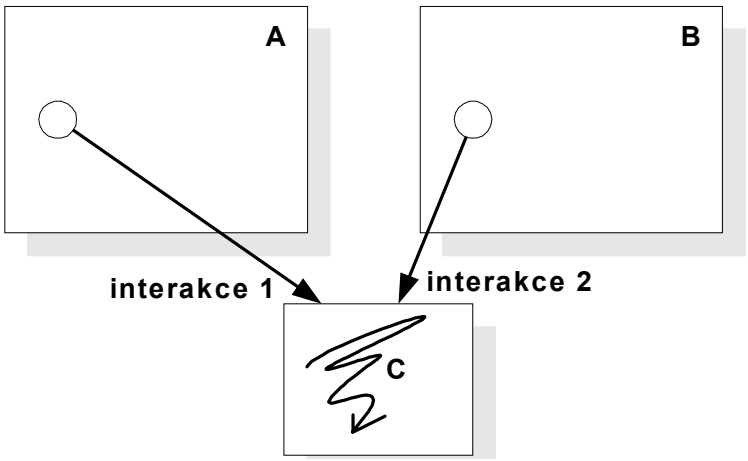
Představme si, že existuje výskyt nějaké (libovolné) informace *A* a existuje výskyt nějaké (libovolné) informace *B*. Zjistíme, že ve výskytu informaci *A* se vyskytuje určitá část z A, která se opakuje i v *B*, označme tuto opakující se část jako *C* (viz následující obrázek, část C označena klikyhákem):



obrázek 7 Situace vedoucí k opětovné použitelnosti (re-use)

Pro zavedení opětovné použitelnosti musí být zavedena interakce mezi prvky typu A, B, atd. Opakující se část *C* se „vytkne“ mimo oba prvky A, B a zavedenou interakcí se prováže zpět do bodů, ze kterých byl prvek C vytknut.

Vznikne tak obdoba odkazu. Výsledkem je následující situace (viz obrázek 8):



obrázek 8 Situace zavádějící opětovnou použitelnost výskyty interakcí

Zavedení opětovné použitelnosti není v žádném případě nějakou novinkou a je obecným a dobře známým jevem v programování, obecněji v modelování a tvorbě SW. Můžeme namátkou jmenovat následující příklady opětovné použitelnosti:

- volání funkcí ve strukturálním programování
- normalizace databáze ve smyslu vytknutí sloupců téhož analytického významu do samostatných tabulek
- dědění a jiné interakce mezi třídami
- obecně interakce mezi prvky modelu v UML, kdy jeden prvek používá druhý prvek (např. include a extends mezi případy užití apod.)
- aj.

Princip maximální opětovné použitelnosti zní v tomto případě tak, že pokud není uvedeno jinak (kdy nastává ono „jinak“ viz dále) je třeba vždy zavádět opětovnou použitelnost v tom smyslu, jak ukazuje obrázek 8. Jinak řečeno, obrázek 7 Situace vedoucí k opětovné použitelnosti (re-use) je chápán jako chybový stav a je třeba provést operaci vytknutí a zpětného provázání do bodů vytknutí. Princip maximální opětovné použitelnosti říká následující: obrázek 7 Situace vedoucí k opětovné použitelnosti (re-use) je považován za indikaci chyby. Části, které by se opakovaly, se nesmí tvořit dvakrát, ale musí se vytknout jako jeden prvek a poté je třeba se na něj z několika míst odkazovat (ukazovat si na něj interakcí). To platí pro libovolnou situaci, která se řeší.

Pro tento princip je důležitá jeho obecnost: Prvky *A*, *B*, *C* mohou být cokoli, čeho se může náš problém týkat. V předešlém odstavci byly uvedeny jako příklady opětovné použitelnosti interakce mezi prvky SW (např. funkce, třídy apod.). Nemusí se však vždy jednat pouze o prvky SW. Principu opětovné použitelnosti by se měly podřídit také jiné dokumenty, jako jsou metodiky firmy nebo dokumenty vznikající při tvorbě SW apod. Znamená to, že prvky v interakci opětovné použitelnosti mohou být opravdu „cokoliv“, tj. vše, co vyžaduje opětovnou použitelnost. Mohou jimi být (a měly by být) samozřejmě prvky modelu SW, tj. prvky vyvíjeného SW, a to až po kód. Navíc však tomuto principu maximální opětovné použitelnosti podléhají například také postupy ve firmě, dokumentace projektu, zavádění vzorů, opakování postupů při designu apod. Princip maximálního re-use přikazuje opakující se postupy nějak vytknout a opětovně je použít, což vede k zavedení postupek, k tvorbě návodek, k zavedení návrhových vzorů apod. Současně se vyžaduje, aby tyto dokumenty (metodiky, vzory apod.) samy mezi sebou dodržovaly princip maximální opětovné použitelnosti, tj. aby se mezi sebou odkazovaly a propojovaly se tak, aby nedocházelo k opakování částí dokumentů. Nic z toho by se nemělo vyskytovat dvakrát, což znamená, že se vždy musí provádět vytknutí a odkaz na prvky.

Jak vidět, princip opětovné použitelnosti patří k velmi silným a přitom jednoduchým návodům, navíc je aplikovatelný všestranně, včetně zavádění technologie postupek do firmy, tvorby dokumentace aj.

Poznámka: Při psaní kódu se velmi často narušuje princip opětovné použitelnosti tzv. „CLIPBOARDOVOU CHYBOU“. Pracovník zjistí, že se určitá část kódu opakuje a tak ji přenese přes schránku („clipboard“). V jedné firmě tuto metodu nazývali také PASTOVÁNÍ.

Jiný způsob, jak systematicky porušovat princip opětovné použitelnosti, je zavést metodu TUNEL spolu s metodou příčného řezu, kdy vývojáři mezi sebou nevědí, co dělají jejich kolegové. V jedné firmě, která vyvíjela poměrně rozsáhlé systémy strukturálním způsobem, se jeden z vedoucích pracovníků rozhodl spočítat ve velmi rozsáhlé knihovně zdrojových kódů, kolikrát je naprogramována jedna nepříliš složitá „převodníková“ funkce. Dospěl k zajímavému výsledku: Napočítal celkem 36 výskytů této funkce v systému. Poté tuto práci vzdal, protože toto číslo mu už stačilo k obrazu, jaký je stav firemní knihovny.

Nedodržení principu maximální opětovné použitelnosti vede k následujícím nepříjemným efektům:

- nastává opakování prací (při vývoji, při tvorbě dokumentace, při tvorbě metodik apod.), poté jako důsledek následuje ztráta efektivity vývoje
- dochází opakování oprav při změnách (což je podobné jako předešlý bod, ale v jiné fázi tvorby a údržby SW)

- nastává velmi nepříjemný efekt - ztráta transparence řešení, což je důsledek vedoucí ke katastrofám. Díky nedodržení principu maximální opětovné použitelnosti se některé věci opakují několikrát a to „bůhví kde“. Věci nejsou na svých logických místech, kde by se měly nacházet. Dokumentace je obtížnější, je nelogická a navíc mnohem více pracná. Hledání všech opakujících se změn je neúnosně zdlouhavé, systém se stává zbytečně složitý a nepřehledný, v důsledku více chybový. K největší ztrátě efektivity dochází právě díky ztrátě transparence, která vzniká jako důsledek nedodržení principu maximálního re-use.

Poznámka jen na okraj: Pokud dojde v předešlém příkladu ke změně v dané funkci, která se minimálně 36krát opakuje, kde je jistota, že tuto změnu promítneme do všech opakujících se výskytů této funkce?

Je dobré v této chvíli upozornit na velmi vážný závěr předešlých úvah: Nedodržování principu re-use vede ke ztrátě transparence systému. Vyplývá z toho, že tento princip má svou obrovskou důležitost pro další naše úvahy.

I zde však existují výjimky: Samotná opětovná použitelnost nemusí být obecně vždy vyžadována a dokonce někdy bývá úmyslně porušována. Uveďme jako příklady: Úmyslné kopírování prvků, optimalizace návrhu, rozpouštění prvků v designu, denormalizace databáze, zdvojení informace kvůli dosažení vyšší rychlosti zpracování apod. Jedná se o kroky, které mají za úkol dosáhnout nějaké technologické výhody (rychlost, paměť), přičemž dochází ke ztrátě opětovné použitelnosti.

2.2 Úplnost dokumentace z hlediska opětovné použitelnosti

Pro použití principu re-use se nejenom že samotný informační systém stává více transparentní, ale také se velmi zjednodušuje systém dokumentace. Efektivní technologie tvorby SW doporučuje použití tohoto principu na samotné zavádění technologie.

Mezi dokumenty projektu musí existovat sjednocující princip re-use a žádný artefakt zavedených postupů se nesmí opakovat. Všechny opakující se situace postupů musejí být vystiženy tak, aby byly dány odkazem a nikoliv opakováním „stejných elementů“ v jednotlivých částech dokumentace znovu a znovu. Použije se metoda vytknutí a odkazu vždy, když se něco má opakovat. Důsledkem toho je například

zavádění metodik (opakující se postupy), zavedení vzorů (odkaz na vzor) apod. Například obrázek 4 Postup mapování je dán katalogem vzorů ukazuje nejenom princip mapování z analytického modelování do designu, ale vyjadřuje také opětovnou použitelnost při použití katalogu mapování pomocí odkazu do číselníku všech mapování.

Protože se jedny prvky dokumentace vždy odkazují na určité jiné prvky dokumentace, tak se celkový systém pro tvůrce velmi zjednoduší. Například u zmíněného obrázku autor nějakého konkrétního mapování do designu v konkrétním projektu nezdokumentuje samotný postup mapování, ale vybírá jej jako vzor a odkazuje se něj jako na již připravený prvek v katalogu. Dokumentace mapování v daném konkrétním projektu je tak velmi jednoduchá, protože používá pouze metodu odkazu. Postup mapování je uložen „někde jinde“ a to v katalogu všech postupů mapování. Designér pouze napíše: „mapování podle tohoto vzoru ..., výsledek viz tento model“. Dokumentace projektu se tak výrazně zjednoduší, zpřehlední a zeštíhlí.

2.3 Chyba ztráty identity prvku

Jedna z velmi častých chyb, která vzniká jak při modelování, tak při tvorbě informačního systému, je *chyba ztráty identity prvku*. Podstata této chyby je následující:

Na obrázku viz *obrázek 7 Situace vedoucí k opětovné použitelnosti (re-use)* je opakující se část C namalována takovým způsobem, aby bylo zřejmé, že se opakovaně vyskytuje v obou prvcích A a B, což vede k myšlence „vytknutí“. Avšak takto namalovaný obrázek je třeba ze zásady vidět jinak, než tak, že „C je obsaženo v obou prvcích“. Protože opakující se část C není identifikována mimo prvky A a B, tak část C jako taková vlastně v modelu neexistuje a na obrázku C nemá smysl o prvku jako takovém ani jako o C hovořit. Z toho důvodu je tato „opakující se část C“ namalována jako neobvyklý klikyhák a nikoliv rovnocenně s A a B jako obdélník. Platí totiž jednoduchá skutečnost, že to, co není v modelu zavedeno jako prvek modelu, tak to neexistuje a nelze o tom hovořit. Je zřejmé, že se nelze odkázat na něco, co je „pouze nějak uvnitř a není identifikováno jako samostatný prvek“. Odkázat se dá pouze na „existující prvek“ a ten je zaveden jako C až v druhém obrázku, nikoliv v prvním. Tento způsob myšlení, kdy „něco jakoby existovalo“, ale jedná se pouze o neidentifikovatelnou část prvku, budeme nazývat chybou ztráty identity prvku.

Při uvedené chybě nastává zvláštní paradoxní situace. Dokonce se dá říci, že se jedná o klasický protimluv: Něco neexistuje a přesto se o tom hovoří (viz například věta opakující se C na uvedeném obrázku). Podobně se může stát, že analytik spolu s konzultantem hovoří o určitém pojmu a tento pojem nakonec nikde v modelech neexistuje! A to je již vážná chyba modelování. Zmíněná úvaha totiž vypadá na první

pohled jako „silně teoretická“, ale ze zkušeností vyplývá, že její nedocenění vede k nejhrubším chybám v návrhu systému a to již v analytickém modelu.

Pro bližší vysvětlení chyby ztráty identity prvku použijeme tento názorný příklad: Každý programátor ví, že nelze zavolat „kus kódu“ uvnitř funkce. Jedna funkce je chápána jako jeden celistvý identifikovatelný prvek a teprve tento prvek - funkce může být zavolán. Všimněme si, že novou funkci zavedeme jako jeden nový prvek v množině všech volatelných funkcí, a to i se svým celým vnitřkem a svým obalem (interfacem funkce). Z toho vyplývá, že pokud se ve dvou funkcích opakuje nějaká část kódu (doslova odstavec) a tato část má být vytknuta ven podle principu opětovné použitelnosti (např. jako odstranění CLIPBOARDOVY chyby), tak musíme definovat novou funkci. Do té doby však existoval pouze „opakující se odstavec“ a nikoliv funkce. Před vytknutím nové funkce bylo o jednu funkci méně a po této operaci vznikla nová funkce. Nedošlo tedy k vytknutí již existující funkce, ale odstavec se změnil na funkci a ta se vytknula pomocí interakce volání funkce.

Programátorům se může jevit tento postup jako triviální, ale tento princip funguje mnohem více obecněji, jenom namísto funkcí je třeba si představit jakýkoliv obecný prvek a místo volání funkcí obecnou interakci mezi prvky.

Tímto přirovnáním s funkcemi s jejím nevolatelným vnitřkem (odstavcem) lze pochopit často opakující se chybu ztráty identity prvku, která se vyskytuje již v analytickém modelování. Jedná se o chybu velmi podstatnou a bohužel velmi častou. Prvek, který chceme nějak použít a je o něm řeč, musí být identifikován (tj. zaveden a definován) jako samostatný prvek v modelu a musí být následně použit pomocí nějaké interakce s jiným prvkem. Nesmí (a ani nemůže) být schován jako neidentifikovaný a nedefinovaný prvek „někde uvnitř“, to je zřejmý protimluv. Tento až primitivní fakt, který je velmi dobře pochopitelný při volání funkcí, si mnohdy tvůrci informačního systému neuvědomují. Při tvorbě IS v rovině analytického modelování tak vznikají nejhrubší chyby.

Obranou proti této chybě je vyčleňovat všechny prvky modelu AM, které mají svůj nárok na život (tj. o kterých je řeč) a zavádět je do modelů AM a poté je interakcemi provazovat s jinými prvky. Tím vznikají požadované struktury prvků modelů.

Klasickými příklady takovýchto chyb jsou následující:

- je sice řeč o evidovaných adresách, ale ty se v modelech AM nenacházejí, existují pouze skupiny atributů v jiných entitách
- je sice řeč o evidovaném klientovi banky, ale ten se v AM nenachází, existují jakési údaje v osobě, vzniká jakýsi „kříženec“ osoby a klienta
- aj.

K vysvětlení, jak tato chyba nejčastěji vzniká, se vrátíme v části výkladu věnované optimalizaci systémů. Problém totiž spočívá právě v postupech optimalizací systémů v designu, zejména v relačních databázích.

Poznámka: Vzato čistě matematicky uvedená chyba by se dala vyjádřit jako existence resp. neexistence prvku v daném prostoru (obdoba grupy). V příkladu s funkcemi bychom se měli vyjádřit úplně přesně takto:

Ve funkcích se opakují obsahy některých odstavců. Všimněme si, že v předešlé větě se hovoří o dvou typech prvků: o funkcích a odstavcích. Opakování odstavců znamená, že dva různé odstavce mají stejný obsah kódu. Vznikají tak kandidáti na operaci opětovné použitelnosti. Tyto dva odstavce se stejným obsahem kódu dávají vzniknout do té doby neexistující nové funkci. Zdůrazněme „do té doby neexistující". Chyba ztráty identity prvku znamená, že aniž by se daný prvek v modelu „zviditelnil" (tj. vytknul ven a provázal interakcí), začne se s ním „slovně" pracovat (opakující odstavce zůstanou na svých místech).

2.4 Vnější a vnitřní pohled na prvek a zapouzdření

S otázkou opětovné použitelnosti souvisí další velmi důležitý pojem pro modelování a návrh systémů a tím je vnější a vnitřní pohled na prvek.

Vraťme se k obrázku viz *obrázek 8* Situace zavádějící opětovnou použitelnost. Prvek A interaguje s prvkem C, odkazuje se na něj a používá jej. Podobně také prvek B používá prvek C. Takovýchto interakcí může být v systému samozřejmě mnohem více.

Z toho vyplývá první sice jednoduchý, ale velmi důležitý závěr: Každý prvek musí být mezi ostatními prvky nějak jednoznačně identifikován a proto musí existovat nějaká technologie „ukazování si" na něj (jinak by prvek A a ani prvek B nemohl použít tentýž prvek C). Prvky v systému musejí být vybaveny nějakými identifikátory resp. ukazateli, které je identifikují oproti jiným prvkům. To umožní, aby prvek A se pomocí tohoto ukazatele ve své vnitřní struktuře „odkázal a použil" prvek C. Podobně díky téže vlastnosti se také prvek B „odkáže a použije" tentýž prvek C. Existují tedy ukazatele (identifikátory) těchto prvků.

Připomene, že naše úvahy jsou nyní v obecné rovině pro libovolné typy prvků. V této chvíli nemáme na mysli ukazatel například ve smyslu ukazatele v paměti, ale obecnější pojem - obecný ukazatel v daném prostředí, který „nějak" ukazuje na daný prvek a tím jej dává k použití jiným prvkům. Ukazatel v paměti je pouze speciální případ realizace takového obecně chápaného ukazatele (konkrétně se jedná o ukazatele pro prvky, které obsazují paměť počítače).

Z obrázku vyplývá další velmi důležitý a jednoduchý závěr: Všimněme si, že použití od A k C anebo analogicky od B k C je směrové. Prvek A používá prvek C a prvek B používá prvek C v daném směru. Provázání těchto prvků interakcemi použití je třeba

vždy chápat jako směrové, tj. vždy ve směru od koho ke komu, jinak řečeno ve směru „kdo koho používá“. Upozorníme na jednoduchý fakt, že ne každá interakce mezi prvky musí být směrová, například vazba mezi tabulkami v SQL je automaticky symetrická bez nějakého významu směru od koho ke komu. Avšak při použití jednoho prvku druhým prvkem podle našeho výkladu opětovné použitelnosti je toto použití jako interakce evidentně směrovou. Pokud by například prvek C pro svou funkcionalitu potřeboval zpětně prvek A, museli bychom zadat další interakci jako použití od C k A.

Další jednoduchý, ale důležitý závěr vyplývá z toho, že pokud prvek A používá prvek C, tak mu prvek C „něco poskytuje“, tj. existuje důvod, proč (a následně samozřejmě jak) prvek A používá prvek C. Jinak by totiž nemělo smysl, aby se prvek A odkazoval na prvek C. Pokud prvek A používá prvek C, tak to jinak řečeno znamená, že prvek A používá služby prvku C. Podobně prvek B používá prvek C tak, že používá jeho (nějaké) služby. Někdy se ekvivalentně namísto služeb hovoří o požadavcích (angl. requests) ze strany prvku A vůči prvku C, který tyto požadavky splňuje, což je synonymum pro poskytování služeb. Protože prvek C je pouze jeden, tak množina všech jeho služeb je stejná ať už je viděna ze strany prvku A nebo ze strany prvku B.

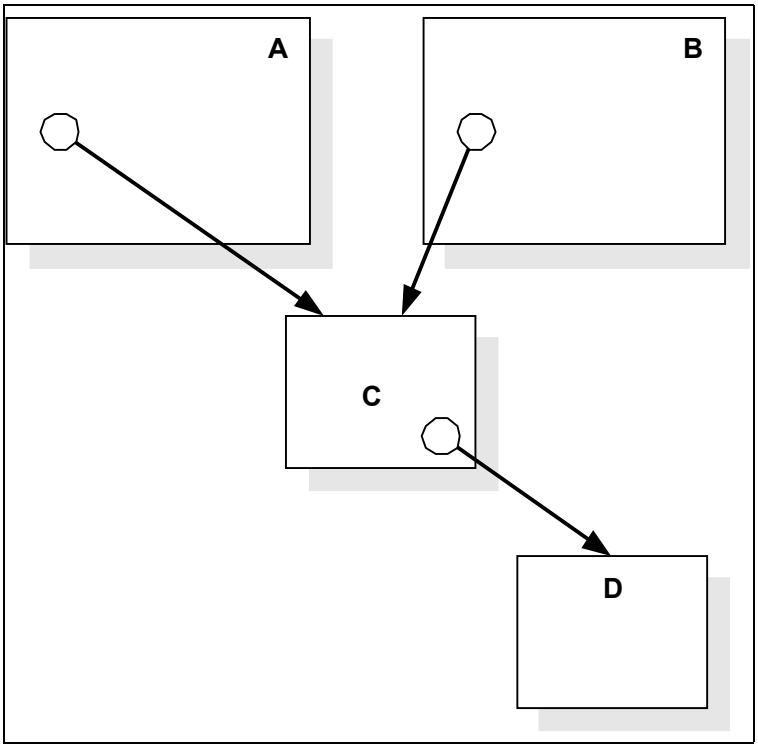
Nyní je třeba trochu vysvětlit, co máme v předešlém odstavci vlastně na mysli oněmi „službami“ resp. ekvivalentně „požadavky“. Je třeba zdůraznit, že naše úvahy jsou nyní velmi obecné a týkají se libovolných typů prvků, které mají možnost zavést mezi sebou princip opětovné použitelnosti. Máme na mysli podobně jako u ukazatelů obecné služby, které jsou v daném prostředí nějak „konkrétně zhmotněny“. Tyto služby jsou vlastně vyjádřením věty „prvek A používá prvek C“, což je synonymum pro „využívá jeho služby“.

Zopakujme si nyní závěry z našich úvah o principu opětovné použitelnosti:

- existují ukazatele prvků (jednoznačné identifikátory prvků), což dává možnost odkazu mezi prvky pro použití
- interakce je směrová ve smyslu jeden prvek používá druhý prvek v daném směru použití
- protože jeden prvek používá druhý prvek, znamená to, že jeden prvek poskytuje „nějaké“ služby druhému prvku, resp. ekvivalentně řečeno, jeden prvek plní „nějaké“ požadavky vznesené od druhého prvku

Nyní po těchto závěrech se dostáváme k velmi důležité základní vlastnosti zmíněné interakce použití, která určuje její „objektovou povahu“. Nejprve je třeba si uvědomit ten prostý fakt, že interakce při použití může být zavedena mezi dvěma libovolnými prvky daného typu A, B, C případně D atd., které ji potřebují. Znamená to, že na jedné straně například prvek C je použit (v našem příkladu prvky A a B), ale on sám

může použít další prvek, označme jej například D. Vzniká tak obdoba rekurzivní povahy této interakce ve smyslu „jsem použit, ale mohu také někoho použít“. Tuto situaci pro prvky A, B, C a D můžeme znázornit např. takto:



obrázek 9 Použitý prvek C používá další prvek D

Předešlý obrázek je velmi důležitý. Spolu s vyjmenovanými body z předešlého odstavce nás zavádí k jednomu z principů objektového paradigmatu, který se nazývá zapouzdření.

Všimněme si, jak prvek A nebo B „vidí“ prvek C, když jej používá. V obou obrázcích, jak na obrázku viz *obrázek 8*, tak na obrázku viz *obrázek 9*, „vidí“ prvek A díky nějakému ukazateli prvek C, používá jeho služby, ale z hlediska vnitřní struktury C „to je pro něj vše“. To, že prvek C používá nebo nepoužívá prvek D je z hlediska interakce použití prvku C prvkem A (resp. prvkem B) nezajímavá skutečnost a dokonce při této interakci použití od A k C není vnitřní struktura C vůbec vidět. Je to důsledek toho faktu, že prvek A potřebuje prvek C k použití (potřebuje jeho ukazatel a služby), tj. ukáže si na něj a použije jeho služby bez ohledu na to, zda prvek C vnitřně používá nějaké další prvky anebo nikoliv. Každý prvek jakoby obsahoval

„vnější obal“, který je reprezentován identifikátorem a službami poskytovanými ven k použití a „vnitřkem“, který reprezentuje vnitřní strukturu daného prvku.

Z toho vyplývá, že díky uvedeným vlastnostem interakce použití se na tento prvek musíme dívat „dvěma různými pohledy“. Buď se jedná o pohled vnější a ten je reprezentován ukazatelem na daný prvek a současně službami, které prvek nabízí ven, anebo se jedná o pohled vnitřní, kdy hovoříme o vnitřní struktuře daného prvku. Jedná se o dva různé pohledy na tentýž prvek, přičemž tyto dva pohledy „nejsou nikdy promíchány“. Vždy se díváme buď jedním nebo druhým pohledem.

Klientem daného prvku budeme nazývat ten prvek, který přes daný ukazatel na prvek používá daný prvek. Na předešlých obrázcích jsou prvky A a B klienty prvku C a prvek C je klientem prvku D.

Z uvedeného vyplývá jeden důležitý fakt: Pokud hovoříme o daném prvku, tak bychom měli (pokud to není jasné z kontextu věci) uvést, který z těchto dvou pohledů máme v té chvíli na mysli, zda vnější anebo vnitřní pohled na daný prvek. Zda se jedná o pohled, jak prvek vidí klient anebo zda chceme pitvat vnitřní strukturu prvku.

Například na předešlém obrázku uvažujme o zmíněných vnějších a vnitřních pohledech na prvek C. Vnějším pohledem na něj je: „C je identifikován a nabídnut k použití“ (takto jej vidí A, B na koncích šipek). Vnitřním pohledem na prvek je „C má vnitřní strukturu“ (C používá D). Všimněme si, že vnitřní pohled na jeden prvek obsahuje vnější pohled na jiný prvek, což je právě podstata „rekurzivní povahy“ interakce použití. Například podobně vnitřní pohled na strukturu prvku A obsahuje vnější pohled na prvek C, stejně, jako vnitřní pohled na prvek B také obsahuje vnější pohled na prvek C.

Vnější pohled budeme také nazývat „interfejsový“ pohled nebo klientský a vnitřnímu pohledu budeme také říkat „implementační“ pohled.

Každý prvek se chová podobně, jako by měl „vnější obal“ k použití. Klient daného prvku tento obal identifikuje přes ukazatel a může jej používat. Až za tímto obalem je schována vnitřní struktura prvku, která je navržena tak, že tyto používané služby obalu „implementuje“, tj. realizuje. Klient však vidí pouze vnější obal služeb a nevidí vnitřní implementaci služeb.

Podstatou paradigmatu zapouzdření je ta skutečnost, že oba pohledy, jak vnější, tak vnitřní, jsou disjunktní v tom smyslu, že při vnějším pohledu vidí klient přes ukazatel pouze daný prvek pouze jako nabídnuté služby (tj. nevidí vnitřek prvku) a tyto služby může používat. Naopak při vnitřním pohledu je v prvku viditelný pouze vnitřek prvku a není vidět venkovního klienta, tj. toho, kdo používá služby. Svět se dělí pro každý prvek na dva pod-světy: Jeden je reprezentován pohledy zvně (vnější svět na straně klienta) a druhý reprezentuje pohled na prvek zevnitř.

„Slupkovitá rekurze vztahu“ spočívá v tom, že vnitřní svět prvku se stává vnějším světem pro jiné prvky, tj. daný prvek se ve své vnitřní struktuře může stát klientem dalších prvků, které jej takto strukturují.

Podobně je tomu s libovolným prvkem modelu. Existuje vnější pohled na libovolný prvek modelu. Tento pohled vyjadřuje jedinou skutečnost: „To je tento identifikovaný prvek, na který lze ukázat a který lze použít“. V tomto pohledu vůbec není obsažena vnitřní struktura prvku, která je reprezentovaná vnitřními interakcemi tohoto prvku.

Použití prvku znamená v úplné obecnosti mít možnost ukázat na tento prvek a žádat jej o něco (služby, požadavky apod.). Druhý pohled na prvek modelu je vnitřní pohled, což je pohled na interakce tohoto prvku s jinými prvky. Tento pohled je však až za hranicí vnějšího pohledu na prvek, tj. je „uvnitř“ tohoto prvku. To je druhý, vnitřní pohled na prvek modelu, na jeho skladbu, pohled na jeho strukturu. Přitom platí uvedený rekurzivní mechanismus: Z hlediska interakce tohoto prvku s dalšími prvky nahlíží daný prvek na tyto své vnitřní další prvky vnějším pohledem a vnitřky jeho vnitřních prvků jsou pro něj opět skryty.

Platí velmi důležitá a praktická rada pro modelování v UML vyplývající z praktických zkušeností: Je třeba vždy tyto dva pohledy na prvek modelu, tj. vnější a vnitřní pohled, vždy od sebe důsledně oddělovat. Buďto se hovoří o prvku A z hlediska vnějšího pohledu, tj. jako o prvku ve smyslu celku, anebo se hovoří o prvku A z hlediska jeho vnitřní struktury, nikdy však současně! Uvedená zásada velmi úzce souvisí s dalším pojmem a tím je „objektově orientovaný přístup“.

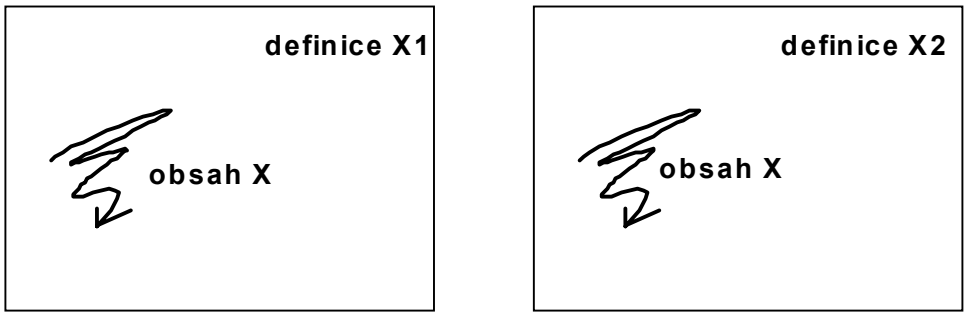
2.5 Třídy a jejich instance

V předešlé kapitole byly vysvětleny základní vlastnosti interakcí, které vedou obecně k možnosti zavést opětovnou použitelnost. Kromě uvedených vlastností charakterizované „vnějším obalem“ a „vnitřní strukturou“ prvků (existence dvojího pohledu na prvky, interfejsový a implementační) je třeba provést ještě jednu úvahu týkající se opětovné použitelnosti. Vysvětlíme si ji pomocí následující situace:

Představme si, že chceme v systému zavést nový prvek, nazvěme jej třeba X1. Je identifikován a má své nějaké vlastnosti. Tyto vlastnosti samozřejmě musíme zavést nějakou definicí tohoto prvku. Znamená to, že tento nový prvek je zaveden nějakou definicí a současně bude nějak používán nějakými jinými prvky.

Nechť po určité chvíli naší tvůrčí práce nad systémem budeme potřebovat další nový prvek, označme jej například jako X2. Je možné, že přitom zjistíme tuto zajímavou shodu: Daný nový prvek X2 bude mít úplně stejné vlastnosti, jako má prvek X1. Tento nový prvek X2 vypadá jako klon již existujícího prvku X1. Z našeho výkladu to znamená, že má sice nový identifikátor (je to nový prvek k novému použití), ale skupina možných služeb, které poskytuje, a jeho vnitřní struktura jsou zavedeny stejně jako u prvku X1. Jedná se o vlastně o další nový prvek stejných vlastností. Z hlediska nového zavedení prvku to znamená opakovat tutéž definici, kterou jsme

již zavedli u X1, také u nového prvku, znovu pro další prvek stejných vlastností. Dospěli jsme k zajímavé a již známé situaci, kterou si znázorníme tímto obrázkem :



obrázek 10 Dvě definice prvků se stejným obsahem

Daná „definice prvků stejných vlastností“ se opakuje úplně stejně, jak to zobrazuje *obrázek 7* Situace vedoucí k opětovné použitelnosti (re-use). I v tomto případě bychom měli zavést opětovnou použitelnost a centralizovat definici prvků stejných vlastností operací vytknutí. Zde tato operace má význam „jsem z dané třídy“.

Princip opětovné použitelnosti aplikovaný na definici prvků vede k efektu zavedení tzv. třídy prvků, nebo budeme také říkat ekvivalentně typ prvků. Úvaha při zavedení třídy prvků resp. typu prvků je velmi jednoduchá: Daná definice prvku se neopakuje a zavede se pouze jednou centrálně a nazve se třída prvků resp. typ prvků. Poté se prvek prohlásí za pocházející z dané třídy, tj. že je daného typu. Tím se tento prvek vlastně odkazuje na třídu (na typ), ze které pochází a tím jsou také dány jeho vlastnosti.

Této vlastnosti, která opět platí obecně pro libovolné prvky, budeme říkat dichotomie tříd a instancí neboli rozdělení na třídy a instance. Dané třídy jsou „centrálními“ vytknutými definicemi prvků, ze kterých dané prvky pocházejí a tím mají dané vlastnosti. Daný prvek pocházející z dané třídy (definovaný danou třídou) budeme také nazývat instance třídy. V daném systému se tedy nacházejí třídy (definice typů) a jejich instance (samotné prvky z těchto tříd). Vztah od instance ke třídě budeme nazývat vztah „meta“.

2.6 Principy objektově orientovaného přístupu

Uvedené poznatky si nyní dáme dohromady do ucelené kapitoly.

Programátoři se většinou seznamují s principy tak zvaného *objektově orientovaného programování*“ ve zkratce nazývané také OOP. Jedná se o principy zavedení objektů v programování, například v jazycích JAVA, C#, DELPHI apod. V programování se používají základní pojmy objektového programování jako jsou objekt, instance, třída, interface, dědění, polymorfismus apod.

Avšak existuje obecnější pojetí objektů, než je objektově orientované programování, a tím je *objektově orientovaný přístup.* Dále budeme pro toto zobecnění používat zkratku OOAP jako *Object Oriented Approach*. Tento přístup je zobecněním objektového přístupu pro všechny prvky, které se podřizují principům vyjmenovaným v předešlé kapitole.

Je třeba podotknout, že tyto dva pojmy objektově orientované programování OOP a objektově orientovaný přístup OOAP je třeba od sebe odlišit, i když spolu úzce souvisejí. Jejich vztah je takový, že pojem objektově orientované programování OOP spadá pod obecnější a pod rozsáhlejší pojem objektově orientovaný přístup OOAP . OOP lze chápat jako zvláštní případ OOAP specializovaný konkrétně na oblast programování.

Existuje několikero možných prostředí, kterým se říká *objektově orientované*.
V každém z takových prostředí má pojem „objekt“ jiný význam. Lze ale vypozorovat určité společné rysy těchto prostředí. Mají společné právě to, co chápeme jako „objektové prostředí“ a jsou to ty vlastnosti, které jsme si popisovali v předešlé kapitole. Jinak řečeno, z hlediska objektově orientovaného přístupu lze vysledovat společné vlastnosti objektů ve všech OOAP prostředích.

Těmito základními vlastnostmi objektově orientovaných prostředí jsou tyto:

- je zaveden vnější a vnitřní pohled na prvky - objekty. Toto rozdělení na dva pohledy má povahu zapouzdření s těmito dvěmi základními vlastnostmi, které činí objekt objektem:
    - Vnější pohled na objekt (tj. užití objektu zvně) nemá přímo zpřístupněnu vnitřní strukturu objektu. Ukázání na objekt v některé z interakcí je ukázání na reprezentaci vnějšího pohledu, tj. na reprezentaci obalu tohoto prvku, na veřejné rozhraní služeb tohoto prvku. Současně s ukázáním na veřejné rozhraní konkrétního objektu , tj. na vnější reprezentaci služeb konkrétního prvku, je tento prvek pochopitelně jednoznačně identifikován oproti jiným prvkům Objekt je identifikován mezi jinými objekty a může být požádán zvně o nějakou službu tohoto rozhraní (obalu). Důležité je, že implementace prvku je při vnějším pohledu, tj. při této žádosti o službu, zvně skryta. Jediné, co vidí žadatel o službu u daného objektu, je množina služeb objektu.
    - Prvek definovaný ve své vnitřní struktuře neví „kdo, kdy a jak“ jej použije, tj. kam a do jaké interakce bude prvek dosazen a kým bude o službu požádán. Vnitřní pohled neví nic o konkrétním vnějším použití

(pohledu zvně), tj. o použití prvku v interakci. Říká se, že použití prvku je z hlediska vnitřního pohledu anonymní. Pro uživatele prvku se zavádí obecný název klient prvku. Klient zůstává z hlediska vnitřního pohledu anonymní. Hovoříme takto ekvivalentně o anonymitě klienta objektu.

- existuje tzv. dichotomie (rozdělení prvků do dvou skupin) na třídu (typ, druh) a instance třídy. Vlastnosti prvků nejsou definovány přímo v daných prvcích, ale pomocí třídy, kam prvky patří. Daný prvek nemá definovány vlastnosti v sobě, ale má své vlastnosti dány díky příslušnosti k třídě (typu) jako její tzv. výskyt neboli instance. V třídě jsou vlastnosti definovány velmi podobně jako v biologii. Instance neboli výskyt patří k dané třídě (pochází z ní) a proto má tato instance takové vlastnosti, které jí tato třída poskytuje ve své definiční části. Je zajímavé, že obecně samy třídy podléhají opět principům OOAP, tj. jsou chápány jako objekty. Lze tedy opět rozlišit vnější a vnitřní pohled na třídu jako objekt (první dvě vlastnosti objektů) a také lze zavést dichotomii u třídy, tj. třídu tříd (druh druhu). Jinak řečeno platí vlastnost dichotomie o existenci druhu na samotný druh. Znamená to, že zkoumaný druh výskytů, tj. třídu, lze chápat opět jako výskyt nějakého vyššího druhu (vztah meta meta). Rekurzivní přechod „meta“ nahoru k druhu druhu, dále k druhu druhu druhu atd. je sice teoreticky neuzavřený, ale zastavuje se na té úrovni, kdy je přechod k dalšímu vyššímu druhu druhu z hlediska řešení konkrétního problému dále nezajímavý a je tedy prakticky zbytečný.

Uvedené vlastnosti jsou v přímém vztahu k principu maximální opětovné použitelnosti. První vlastnost zavádí možnost, aby N klientů mohlo bez kolizí použít tentýž výskyt (ukázat si na něj a použít jeho obal). Navíc se umožňuje, aby použitý prvek rekurzivně používal jiné výskyty jako klient dalších výskytů (použití vnitřní struktury podléhá opět principům OOAP). „Druhová vlastnost“ umožňuje neopakovaně definovat vlastnosti výskytů stejných vlastností v jednom místě, tj. ve třídě. Současně je třeba upozornit na jednoznačnou identifikaci prvku jako identifikaci jeho vnějšího obalu a nikoliv identifikace jeho vnitřní struktury, která je až za obalem zvně neviditelná.

Je třeba upozornit na dva základní průvodní jevy zapouzdření v obecné rovině OOAP, a tím jsou tyto skutečnosti:

- při použití objektu je vidět obálka – služby a nikoliv vnitřní implementace objektu
- v implementaci objektu není vidět klient, který zůstává pro vnitřek objektu anonymním

Následuje výčet těch objektově orientovaných prostředí, která splňují uvedené vlastnosti OOAP a přitom jsou pro tvorbu IS důležitými.

- objektově orientované programování

- analytické modelování
- modelování podniku
- syntaxe UML
- lidská mysl obecně, tj. „zdravý selský rozum“

V dalších kapitolách je stručně vysvětleno, co je v těchto prostředích chápáno jako objekt, co jako třída, co je vnější a co je vnitřní pohled.

2.7 Objektově orientované programování

Zde jsou objekty přímo naprogramované struktury v programu jako instance tříd s interfacem. Princip vnějšího a vnitřního pohledu je realizován pomocí zapouzdření se základními pojmy „interface objektu a implementace objektu“. Existuje klientský pohled na objekt: Klient tj. uživatel objektu jako okolní část programu, vidí pouze interface daného objektu, vnitřek nevidí. Existuje druhý implementační pohled na vnitřní strukturu objektu, tj. existuje konkrétní vyplnění kódu za interfacem. Narušení zapouzdření znamená, že klient může pracovat s vnitřní strukturou objektu, aniž by použil interface (například jako public atribut apod.). Pravidla prostředí jsou dána syntaxí daného objektově orientovaného jazyka.

Velmi často se v literatuře uvádějí jako tři základní axiomatické vlastnosti OOP tyto:

- Zapouzdření
- Polymorfismus
- Dědičnost

Tyto tří vlastnosti souvisejí úzce s uvedenými principy OOAP a vyplývají z nich:

Zapouzdření v OOP je fyzickým vyjádřením existence vnějšího a vnitřního pohledu na objekt. Z jedné strany lze objekt používat (volat jeho operace), přičemž zapouzdření v OOP skrývá implementaci objektu, z druhé strany je objekt vnitřně implementován (kód metody) s neznalostí, kdo jej používá (volá).

Polymorfismus je vyjádřením speciální situace, kdy mohou existovat dva objekty se stejným interfacem (klient vidí stejnou množinu služeb), ale u každého z objektů jsou tyto služby implementovány jinak. Klient vidí stejné obálky, ale chování objektů je vnitřně různé. Z vnějšího pohledu klient tyto dva objekty nerozlišuje (mají stejný vnější „interfejsový“ pohled), ale vnitřně se objekty liší, mají různou implementaci.

Dědičnost zavádí interakci použití na úrovni tříd, tj. v meta úrovni (nikoliv mezi objekty) ještě před tvorbou instancí. To umožňuje, aby jedna třída použila vlastnosti

definované v druhé třídě (blíže viz interakce GENERALISATION – SPECIALISATION v modelu tříd).

2.8 Analytické modelování

Zde jsou objekty v obecném slova smyslu chápány jako výskyty evidovaných informací v informačním systému. Hranice výskytů informací jsou přesně dány pojmenováním těchto výskytů (tato evidovaná fyzická osoba, tato faktura). Vnitřní struktura instancí informace odpovídá interakcím mezi informacemi (rodné číslo fyzické osoby jako její atribut, dodavatel faktury jako běžná asociace apod.). Vlastnosti těchto výskytů se definují ve třídách informací, čímž se zavádějí pojmy v AM, neboli tzv. analytické třídy (analysis class, entity, logical class apod.). Pravidla pro práci v tomto OOAP prostředí jsou dána syntaxí UML.

2.9 Modelování podniku

Zde jsou objekty chápány jako prvky modelu podniku (pracovník, oběžník, oddělení, služba, proces zpracování XY apod.). Pravidla pro modelování tohoto OOAP prostředí jsou dána syntaxí UML (ale je možné použít i jiný modelovací jazyk).

2.10 Syntaxe UML

Sama pravidla modelování, tj. prostředí práce s prvky modelu v UML, lze také chápat jako objektově orientované prostředí, protože zavedené prvky modelu mají samy o sobě charakter objektů (samozřejmě ve smyslu zásad OOAP, nikoliv OOP). U prvků UML lze určit pravidla pro práci s instancemi, jako jsou například: jak je možné prvky modelu instanciovat a z jakých možných tříd (typy prvků v UML), jaké mají prvky mezi sebou povolené vztahy, jakou kardinalitu, kdo koho může použít, kdo je čí atribut, jakou mají asociaci apod. Toto prostředí se nazývá meta-model UML a pravidla pro práci v něm jsou dána opět syntaxí UML (CLASS MODEL pro meta-model UML). UML je schopno díky této vlastnosti popsat svá syntaktická pravidla pomocí sebe sama (tzv. self-describing).

2.11 Lidská mysl

Není bez zajímavosti, že naše úvahy a myšlení podléhají uvedeným principům OOAP, aniž bychom si to uvědomovali. Vymezení pojmů a jejich instancí v naší abstrakci splňují logicky tyto principy. Pokud pojmenujeme nějakou konkrétní věc, tak o této věci hovoříme z hlediska vnějšího pohledu automaticky jako o „celku“. Například ukážeme: „Tady stojí moje auto...“ Máme tím na mysli pod jedním „prstem - ukazatelem“ jakýsi „sumarizující“ pohled, protože při tomto ukázání na tuto věc máme na mysli automaticky „moje auto se vším všudy...“ Jedním ukázáním na tuto věc jsme vymezili všechny hranice tohoto pojmu a současně jej tak i „pojmově zapouzdřili“ (moje auto se vším všudy). Na druhou stranu pokud začneme zkoumat „vnitřní interakce “ tohoto auta, můžeme nalézt spoustu dalších interních věcí, jako jsou jeho vnitřní součástky, jeho zaplacená pojistka a jiné konkrétní pojmy. Navíc tyto konkrétní výskyty jsou instancemi tříd, tj. obecných pojmů (auto jako pojem a toto konkrétní auto).

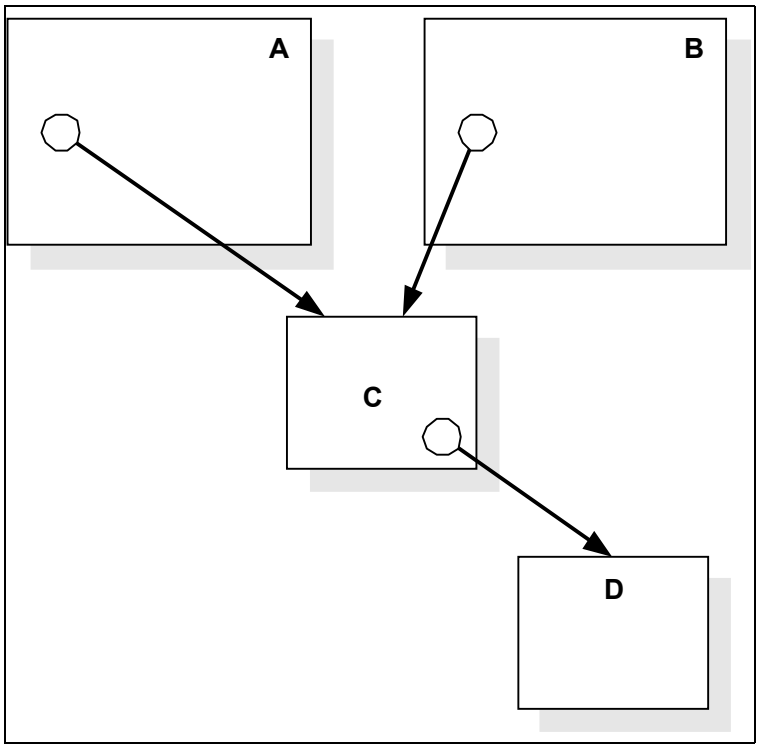
2.12 Objektový přístup OOAP a rozložení diagramů v UML

Spolu s objektovým přístupem OOAP souvisí ještě jeden velmi důležitý efekt související s chápáním diagramů v UML anebo obecně s chápáním libovolného diagramu modelu.

Diagramem obecně rozumíme grafické zobrazení části modelu. Můžeme si představit jeden diagram jako jedno „okénko“, pomocí něhož je nám dán k dispozici grafický náhled na určitou část modelu. Každý diagram obsahuje určité prezentační prvky, tj. zobrazovací prvky daných prvků modelu, které zobrazují část modelu.

Jak vidět, rozlišujeme prvky modelu a prezentační prvky na diagramu. Prezentační prvky na diagramu zobrazují dané prvky modelu, které jsou jakoby „schovány“ za těmito prvky. Jeden a tentýž prvek v modelu může být zobrazen na jednom nebo vícero diagramech, tj. jeho zobrazení se může opakovat.

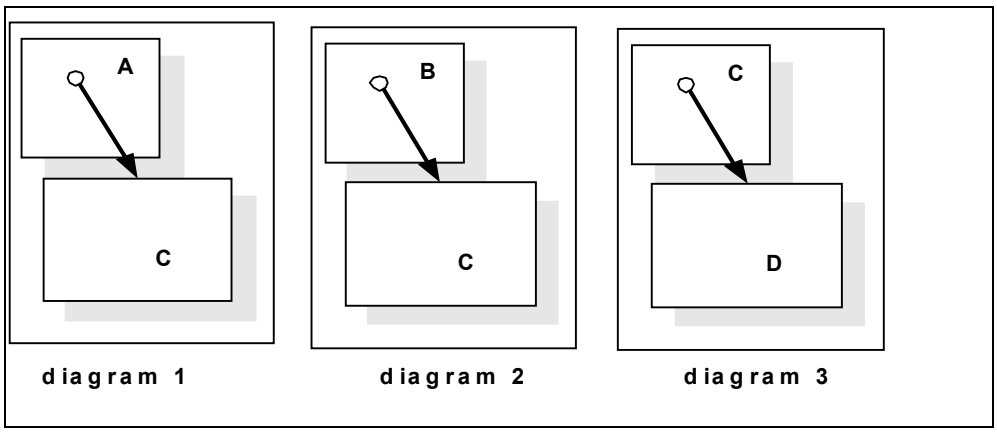
Při pohledu na diagramy je třeba chápat velmi přesně důsledky principů OOAP, které se zde projevují velmi zajímavým efektem, který bychom mohli nazvat jako „dekompozice myšlenky“. Tuto situaci si vysvětlíme si na jednom z již použitých obrázků:



obrázek 11 Znovu zobrazené interakce mezi prvky A, B, C a D

Jenom pro pořádek zopakujme, co nám tento diagram sděluje: Prvek A používá prvek C, prvek B používá prvek C a prvek C používá prvek D.

Z hlediska modelovacích technik (tj. i v UML) je tento diagram úplně ekvivalentní, pokud bychom jej rozvrhli do tří diagramů:



obrázek 12 Tři diagramy rovnocenné s předešlým obrázkem

anebo do jiné kombinace pouze dvou diagramů, např. v jednom diagramu by byla interakce A používá C a B používá C, v druhém diagramu by byla interakce C používá D (případně ještě další možnosti kombinací).

Pokud se zamyslíme nad trojicí diagramů na předešlém obrázku a posoudíme je z hlediska OOAP, tak je zřejmé, že každý diagram (buď 1, 2, nebo 3) je na jedné straně samostatný v tom smyslu, že „něco vyjadřuje“, ale celý model je dán až všemi diagramy všech zobrazení, v uvedeném příkladu až všemi třemi diagramy. Všimněme si, že diagram 1 zobrazuje větu: A používá C a z tohoto hlediska jsou další myšlenky vyjádřené v dalších diagramech (B používá C a C používá D) až „někde dále“ a pro diagram 1 nezajímavými. Pokud však chceme znát vše, musíme „nějak“ projít celý model.

Domysleme tuto myšlenku až do konce: Nemělo by nás překvapit, pokud bychom na dalším zde neuvedeném diagramu (označme jej jako diagram 4) zjistili, že existuje ještě další prvek E (například) a ten je používán prvkem B (například). Tento diagram 4 však musí být také v projektu obsažen.

Vyplývají z toho jednak dvě praktické rady a jedno velmi důležité doporučení pro správný přístup v modelování.

První praktická rada zní: Protože až teprve všechny diagramy dávají celkový obraz o modelu, tak potom pokud chcete o daném prvku vědět opravdu vše, nesmíte se soustředit pouze na diagramy. V tom případě raději použijte prohlížení modelu například reportem apod. Stručně řečeno: Při zkoumání určitého prvku nepoužívejte pouze brouzdání přes diagramy, ale reporty přes projekt.

Druhá rada zní: Na diagramech nepoužívejte příliš mnoho prvků, stává se to velmi nepřehledné. V literatuře se uvádí pravidlo maximálně 8 prvků na jednom diagramu, ale raději bych doporučoval pravidlo jiné: Každý diagram vyjadřuje nějakou myšlenku, nemělo bych jich být víc než 2-3 na jednom diagramu.

Tento postup rozložení diagramů na více s menším počtem prvků a interakcí a tedy s menším počtem myšlenek umožňuje právě přístup OOAP k modelování.

Poznámka: V jedné firmě, kde jsem měl možnost působit, se modeloval systém za pomocí strukturální analýzy. Jeden diagram ERD vyvíjeného systému obsahoval řádově až tisíc tabulek. Diagram se jednou týdně vytisknul na několik archů A2, sekretářky jej skládaly a lepily dohromady na jednu plachtu a poté se pověsil v jedné velké místnosti na stěnu. Tomuto archu se přezdívalo „Česká Třebová“, protože pohled na něj opravdu z dálky připomínal obrovské kolejiště nádražního uzlu.

Poslední, ale nejdůležitější základní doporučení pro správné pochopení principů práce s diagramy v UML vychází právě z pochopení principů OOAP: Každý modelovaný prvek se chová „slupkovitě“, tj. každý jiný prvek, který jej použije v interakci, jej automaticky chápe jako celek s vnějším obalem bez ohledu na jeho vnitřní strukturu. Znamená to, že to co vidíme na daném jednom vybraném diagramu, možná je a možná není vše, co se o daném prvku můžeme dozvědět (viz tři diagramy na předešlém obrázku). Na první pohled nás to může znejistit: Co tedy z diagramu víme, když nevíme vše? Odpověď je jednoduchá: Diagram nám sděluje nějakou „částečnou a relativně malou myšlenku“ celého složitého systému. Celý systém se skládá z velmi mnoha takovýchto myšlenek a OOAP nám zaručuje, že se tyto malé myšlenky dají skládat. To nás dokonce může v určitém smyslu i uklidnit! Každá složitá věc, která je správně chápána, se skládá z několika jednoduchých myšlenek. Ono by to totiž nemuselo takto platit, pokud by neplatilo OOAP!

Vraťme se k našemu příkladu se třemi diagramy na předešlém obrázku. Představme si, tyto diagramy vznikaly v čase. Vyvíjíme a modelujeme systém a v určitém okamžiku naší práce zjistíme, že prvek A používá prvek C (viz diagram 1 na předešlém obrázku). Namodelujeme to a zapíšeme si diagram 1 jako výsledek své zatím neukončené práce. Pokračujeme dále a po určitém čase zjistíme, že prvek C používá prvek D (viz diagram 3). Protože platí OOAP, tak diagram 1 nemusíme zahazovat. Pokud by totiž neplatilo, potom by nové zjištění, že prvek C používá D nějak ovlivnilo diagram 1. Pokud by totiž prvek A používal C a přitom toto použití by znalo a opíralo se o vnitřní strukturu prvku C, potom bychom museli vše z gruntu překopat a diagram 1 by ztratil smysl.

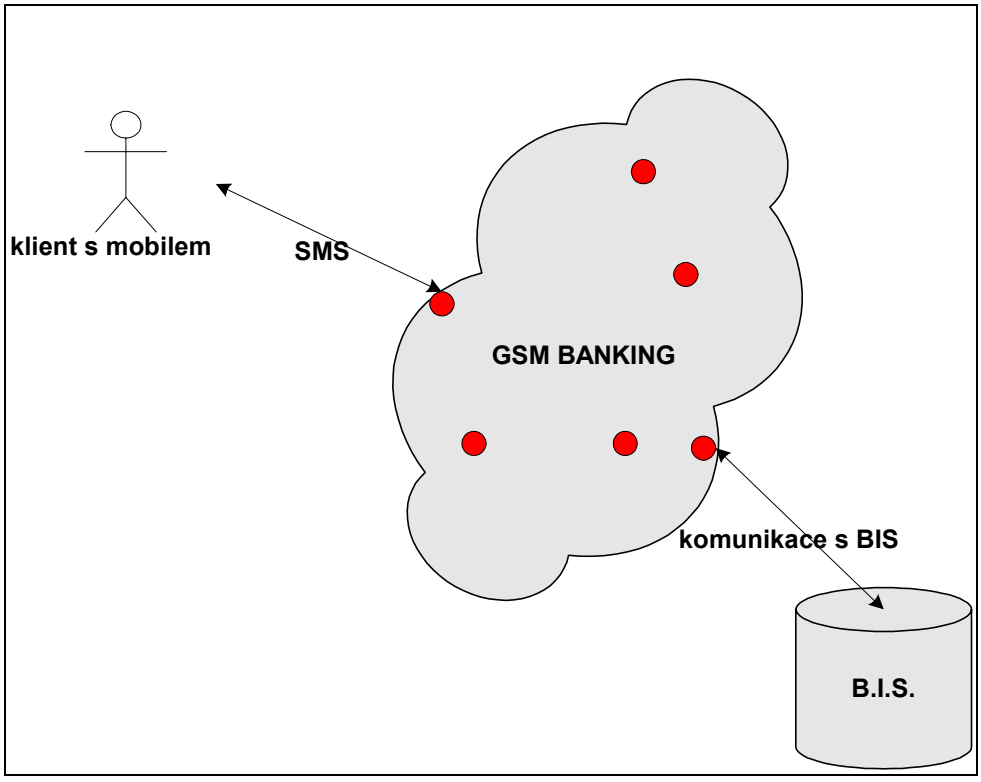
Principy OOAP nám umožňují pokračovat v úvahách a vývoji dále na základě již zavedených prvků, které mohou postupně získat další a další vnitřní strukturu, což v daných modelech již zavedené prvky vně daného měnícího se prvku nepocítí.

2.13 Příklad na anonymitu klienta – monitoring

Poznámka: U tohoto příkladu a následných v této kapitole je potřebná znalost základů objektově orientovaného programování

Tento příklad se skutečně stal, pouze je upraven pro účely našeho výkladu.
V jednom projektu, kde jsem působil jako hlavní analytik, jsme jako tým dostali za úkol vytvořit informační systém tzv. GSM Banking, tj. agendu bankovních služeb přes mobilní telefon a SMS zprávy. V době, kdy probíhaly intenzivně analytické práce (doslova práce na USE CASE MODELU), tak jedna nejmenovaná slovenská banka projevila zájem o tento produkt a stala se tak potenciálním odběratelem. Byly s ní zahájeny rozhovory na analytické úrovni.

Zástupci banky při jednom z rozhovorů o funkcionalitách vznesli následující požadavek: *„Takhle nějak si představujeme váš GSM Banking (viz tento obrázek):*



obrázek 13 Náčrtek GSM Bankingu

Z jedné strany od klienta do vašeho systému přicházejí zprávy SMS a systém mu odpovídá. Z druhé strany váš systém komunikuje s naší bankou, tady na obrázku je náš systém označen zkratkou jako B.I.S. – bankovní informační systém. Potřebovali bychom, abyste jste nám spolu s dalšími funkcionalitami GSM Bankingu dodali i agendu, nazvěme ji třeba Monitoring, která bude sledovat průchod SMS zpráv systémem. V určitých bodech (označeno na obrázku jako červená kolečka) se bude měřit kolik zpráv projde za určitou dobu těmito body. Někde bude obrazovka, tam se našemu pracovníkovi zobrazí měření v těchto bodech. Určený pracovník si může tyto body prohlížet a sledovat, co se v nich v čase dělo. Rádi bychom totiž sledovali například takové věci, jako která vstupní brána SMS je více kdy vytížena, kolik projde zpráv do našeho systému B.I.S. apod. Z toho budeme usuzovat, které části systému posílit, jinak řečeno, chceme vědět, jak systém škálovat. Můžete nám takovou agendu Monitoring dodat?"

Odpověď zněla: *„Samozřejmě…"* (a v duchu s dovětkem, když se za to zaplatí, proč ne ☺).

Na to se ozval druhý kolega z banky: *„Když tak hovoříme o agendě Monitoring, mohlo by se navíc ještě měřit také nejenom v daném bodu, ale také by se mělo udat co se tam měří podle typu služby. Například v některých bodech se bude měřit všechno (všechny služby), v jiných jenom kolik prošlo dotazů na zůstatek účtu, kolik na výpisy a tak podobně. To by určitě zajímalo marketing naší banky, který z toho usoudí, kterou službu posílit a kterou například úplně zrušit apod. Můžete nám takovou agendu Monitoring s měřením průchodů body a jaké služby se tam měří, dodat?"*

Odpověď opět zněla: *„Samozřejmě…"* (a opět v duchu s dovětkem, když se za to zaplatí, proč ne ☺).

Ale obratem jsme vznesli i my požadavek z naší strany: *„Potřebujeme pro chod této agendu znát ty body, kde chcete průchod zpráv měřit. Tam musíme vložit nějaký kód, třebas jenom vyvolání události, ale něco tam být musí. Můžete nám tyto body sdělit?"*

Odpověď zněla: *„Nemůžeme. To ještě nevíme!"*

Podobně pokračujeme v dotazu: *„A také musíme znát, co chcete v těch bodech měřit! Můžete nám to sdělit?"*

A, jak asi tušíte, odpověď zněla velmi podobně: *„Nemůžeme. To taky ještě nevíme!"*

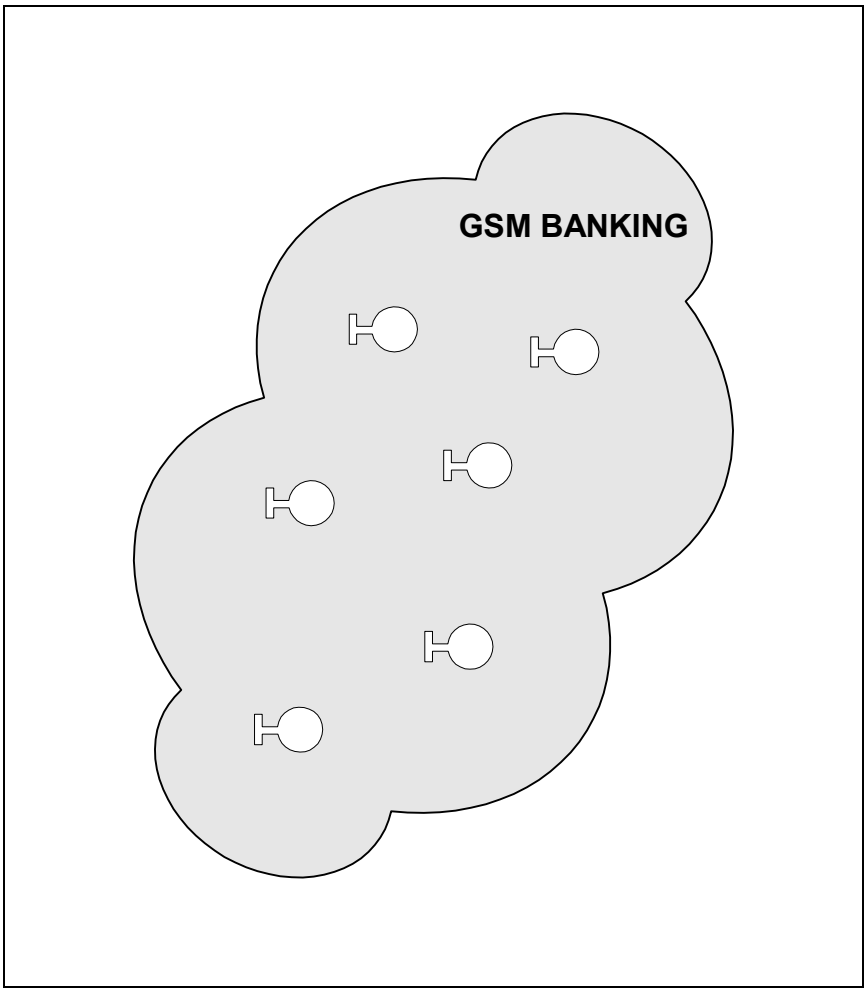
Jak vidět, popsaná situace není pouze příkladem na anonymitu klienta v OOAP, ale ukazuje velmi výstižně, jakou podobu mají rozhovory s budoucími uživateli.

Základní myšlenka tohoto příkladu přichází nyní a formuluje se do této otázky: Je možné při těchto znalostech (lépe řečeno neznalostech) agendu naprogramovat?

(Podotknu jako poznámku, že v jedné firmě, kde jsme tento příklad probírali, mi odpověděli žertem: *„To ještě nevíme ☺.“*)

Odpověď zní: Ano, dá se to naprogramovat a to dokonce tak, že agenda bude i otestována. V dané případě asi tři týdny před implementací banka dodala tzv. konfigurační listy, kde byly uvedeny body „kde měřit“ a „co tam měřit“. Následně se v určitých místech program otevřel, vložil se do něj kód a agenda začala fungovat. Začněme tedy i my řešit tento problém s tím, že jednom okamžiku výkladu upozorníme na jev zvaný anonymita klienta.

Vrátíme se k obrázku, který namalovali zástupci banky a trochu jej rozebereme. V některých bodech daného informačního systému bude „kód otevřen“, do těchto míst se vloží nový kód, který nějak reprezentuje funkcionalitu naší nové agendy Monitoring. Naskýtá se otázka, co bude do těchto bodů vlastně vloženo. Protože náš výklad a celá tato kniha je věnována objektovému přístupu, do těchto bodů budou vloženy objekty ve smyslu OOP (JAVA, DOT NET, DELPHI apod.). Z původního obrázku tak vznikne obrázek nový:



obrázek 14 Nový náčrtek agendy Monitoring s vloženými objekty

Poznámka: Je třeba na tomto místě podotknout, že uvedený problém by dostatečně řešil i strukturální přístup, protože se jedná o velmi jednoduchou agendu. Namísto volání operací objektů by se do daných bodů vložilo volání funkcí.

Nejprve musíme zodpovědět první velmi důležitou otázku: Budou mít tyto objekty v různých bodech stejné vlastnosti, nebo se budou v každém bodě lišit? Řečeno jinak a vyjádřeno programátorsky v OOP: Budou to instance ze stejné třídy, anebo budeme muset pro každý nový bod znovu zavádět novou třídu? Odpověď v tomto případě zní: Všude se měří stejným algoritmem, všechny objekty budou instance z téže třídy a mají proto stejné vlastnosti (pozor, nikoliv informační obsah!). Tímto jsme si tak trochu oddechli, protože představa, že bychom pro každý nový bod

museli zavádět novou třídu, je hrůzná. Vyplynulo by z toho, že pro každý nový bod bychom museli znovu a znovu programovat nový kód.

Druhá neméně důležitá otázka zní: Co ty instance tam vlastně budou dělat? Jinak formulováno: Jaká je jejich hlavní činnost, tj. jak vystihnout stručně a jednoduše činnost, na jejímž základě bude agenda Monitoring vlastně fungovat?

Nabízí se využít názvu agendu Monitoring, ale ukazuje se, že vystihnout činnost pouze slovy „v tomto bodě se monitoruje“, nestačí. V daném bodě dochází k mnohem konkrétnější činnosti. Pokud si odmyslíme všechny možné podpůrné aktivity (tzv. „tanečky okolo“, jako je inicializace, zahájení měření, ukončení měření apod.), tak v daném bodě se vlastně „počítají prošlé zprávy“. V určitém okamžiku se součet vzniklý za časový interval odloží a pokračuje se dále v počítání. Nazvěme proto tento objekt jako Počítadlo.

Nyní přichází zmíněná myšlenka týkající se anonymity klienta: Personifikujme počítadlo a představme si jej jako „tvora“. Ví tento tvor, tj. dané počítadlo, kdy si má zvednout hodnotu počtu zpráv? Nikoliv, neví. A kdo to ví, když ne on? Ví to jeho okolí, tj. klient. Dané počítadlo tedy dostane pokyn zvně, aby si zvedlo svůj mezisoučet. Můžeme si tuto situaci znázornit pomocí pseudokódu tak, že dané počítadlo bude mít operaci zvednutí součtu, které se bude volat například takto:

```
MojePocitadlo.Inc();
```

Anonymita klienta se v tomto případě projevila zatím velmi triviálním, ale účinným způsobem: Je to anonymní klient mimo počítadlo, kdo „ví“, kdy zvednout hodnotu součtu (například v nějaké přesně definované větvi programu). Tím se identifikovala první operace objektu, operace zvednutí **`Inc()`**. Klientem mimo počítadlo je „kdokoliv“, kdo používá počítadlo (doslova z hlediska počítadla „bůhvíkdo“), ale je to ten, kdo „ví“, že teď je třeba zvednout počet.

Podobně můžeme v úvahách pokračovat dále. Představme si, že dané počítadlo v určitém okamžiku běhu programu jako objekt vznikne pomocí volání konstruktoru. Náš nový „tvor“ se narodí „jako hloupé telátko“ ještě někde před voláním zmíněné operace **`Inc()`**. Ptáme se: Ví tento „tvor“, kde, v kterém bodě programu se narodil, tj. ví, kam byl dosazen? Nikoliv, neví, tj. nemůže vědět, kde vlastně měří. A kdo to ví? Odpověď je stejná, jako v předešlém případě: Ví to okolí, tedy klient tohoto objektu a je na něm, aby to našemu počítadlu sdělil. Řešení může být velmi jednoduché: Když banka dodala seznam bodů, kde chce měřit počty zpráv, je třeba tyto body „označit“, doslova „oidéčkovat“, dát jim identifikátory. Vznikne tak konfigurační list, například v této podobě:

```
ID      text
1       vstup brána 1
2       vstup brána 2
3       vstup BIS
4       vstup do autentikačního serveru
```

apod.

Tímto vlastně vznikla určitá dohoda o bodech.

Designér spolu s analytikem určí, kde konkrétně v programu se nacházejí tyto body. Doslova se najde řádek programu, kde se otevře program a vloží se kód.

Buď těsně po zrodu anebo přímo v konstruktoru danému počítadlu můžeme sdělit, kde vlastně měří. Například v bodě 3 (vstup BIS v předešlém výčtu) jednoduše takto:

```
MojePocitadlo = new CPocitadlo();
MojePocitadlo.SetBod(3);
```

Anebo přímo v konstruktoru:

```
MojePocitadlo = new CPocitadlo(bod=3);
```

Podobně bychom zjistili, že můžeme počítadlu sdělit, co vlastně měří podle typu služeb, tj. v bodě 3 a typ služby 6 takto

```
MojePocitadlo = new Cpocitadlo();
MojePocitadlo.SetBod(3);
MojePocitadlo.SetSluzba(6);
```

Samozřejmě je možné i obdobné řešení s jedním voláním např. takto:

```
MojePocitadlo = new Cpocitadlo(bod=3,sluzba=6);
```

Nakonec by bylo možné úvahu ještě více zobecnit. Obecnější počítadlo by nemělo znát ani bod a ani typ služeb, ale dostane identifikaci „předmětu počítání“. V tomto případě je předmětem počítání kombinace dvou hodnot „kde a co“ jako speciální případ předmětu počítání. Nakonec to může vést k zavedení jediné hodnoty „ID počítadla“, které se musí počítadlu od okolí předat v okamžiku zrodu. Jako příklad takového volání můžeme uvést následující:

```
MojePocitadlo = new Cpocitadlo(předmět = 18);
```

Kde jako příklad hodnota 18 odpovídá předešlé kombinaci 3 a 6.

Jak bude tato agenda pracovat jako celek? V běžících programech (komponentách) se budou v přesně definovaných bodech rodit počítadla, budou počítat a v pravidelných (nejlépe nastavitelných) intervalech odesílat počty do databáze i s hodnotou ID svého počítadla. K danému počítadlu tak vzniknou záznamy počtů. Z druhé strany bude fungovat jednoduchá prohlížečka dat, kde si uživatel může prohlížet data těchto počítadel: Zobrazí se mu texty kde a co, a mezisoučty v časových intervalech. Vytištěný výstup z takové agendy pak může vypadat nějak takto:

```
Počítadlo číslo          12
kde:                     vstup BIS
co:                      všechny zprávy
čas zahájení měření      15:30
interval součtování      5 minut
výsledky měření:
čas            počet
15:35          355
15:40          405
15:45          210
```

apod.

Jak se v tomto příkladu projevila anonymita klienta? V konkrétním příkladu agendy Monitoring zná kontext „kde, co a kdy“ anonymní klient, který naše počítadlo používá. Anonymní klient tyto informace předává počítadlu, což vedlo k identifikaci operací počítadla.

2.14 Příklad na anonymitu klienta – objekt na zpracování souborů

V jednom mailu jsem obdržel od jednoho klienta, absolventa školení OOP a UML, velmi zajímavý dotaz. Zajímavé na něm je to, že na něj lze přímo odpovědět pomocí vlastnosti anonymity klienta.

Dotaz byl zformulován takto:

…Vyvíjím jednu drobnou aplikaci, vlastně objekt na zpracování souborů. Tento objekt bude poskytovat několik možných operací pro zpracování souboru (komprese, šifrování apod.). Narazil jsem však na tento problém: V některých případech může operace objektu při zpracování tzv. zkolabovat, může nahlásit error. Požaduji, aby uživatel tohoto objektu nemohl pokračovat ve zpracování souboru dále. Jsem sice schopen tuto chybu ošetřit, dokonce jsem schopen ji sdělit uživateli tohoto objektu, ale mám problém: Jak zabezpečit, aby uživatel dále nepokračoval dalšími operacemi, když nastal error a další operace nemají smysl? Je vůbec možné uživatele objektu donutit k tomu, aby error zpracoval a reagoval na něj tak, že například nepokračuje dál ve zpracování souboru?

Odpověď zní: Nelze jej nutit, klient je anonymní a mimo kontrolu objektu. Dokonce v tom spočívá podstata správně navrženého „chytrého“ objektu. Uživatel jako anonymní klient může kdykoliv zavolat libovolnou dostupnou operaci. Může tak učinit kdykoliv a tedy i velmi „nevhodně“. Podstatnou myšlenkou návrhu správného chytrého objektu je právě tato skutečnost „libovůle anonymního klienta“, se kterou v návrhu počítá. Objekt by měl být schopen vypořádat se s „nepříhodnými“ voláními a to díky správně postavené konstrukci vnitřních stavů. Pokud nastal při zpracování nějaký error, objekt by se měl vnitřně dostat do takového stavu, že poté reaguje na určitá volání buď vyvoláním výjimky anebo návratovou hodnotou konstanty jako error. Klient nemá zakázáno volat tyto operace, ale objekt to „ustojí“.

Všimněme si, že pokud bychom narušili zapouzdření, objekt ztrácí kontrolu nad aktivitami nad souborem a v tomto případě již nelze fyzicky zabezpečit „správnost“ chodu softwaru na zpracování souboru.

Na závěr tohoto příkladu ještě podotknu, že při návrhu takového objektu se s největší pravděpodobností použijí některé vzory z oblasti DESIGN PATTERNS (návrhové vzory), zejména vzor DECORATOR je žhavým kandidátem na použití.

2.15 Příklad na anonymitu klienta – aplikace vláčky

Jeden další absolvent školení mi poslal mailem následující dotaz:

„ …Jako chlapec jsem si vždy velmi rád hrál s vláčky. Nyní po absolvování Vašeho školení OOP a UML vyvíjím pro vlastní potěchu jednu zajímavou aplikaci v jazyce JAVA a to pokud možno objektově. Tato aplikace simuluje hru s vláčky na obrazovce. Mám tento problém: Zatím jsem aplikaci napsal tak, že se vždy maluje na plátno (Canvas) okna formuláře, což je v programu dáno „natvrdo“. Chtěl bych, aby se malovalo na libovolný Canvas, například jaký si určí výběrem obsluha. Nechci, aby se malovalo natvrdo přímo na Canvas konkrétního formuláře. A zde se mi vývoj zadrhl. Jak mohu malovat na nějaké plátno, když nevím, které to vlastně bude?“

Odpověď zní opět stejně: Řešení je v anonymitě klienta. Když daný objekt (v tomto případě část aplikace ovládající hru s vláčky a malující na Canvas) „neví“, kam malovat, tak to ví její okolí a musí jí to sdělit. Existuje několik možných řešení:

První možné řešení:

Každá operace objektů související s namalováním dostává jako vstupní parametr plátno Canvas, například nějak takto:

```
MujObjekt.Draw(gCanvas);
```

Takto uživatel předává objekt `gCanvas` s každým voláním operace namalování doslova podle principu „namaluj se sem, na toto plátno“.

Druhou možností je, že každý objekt, který se má namalovat, má property Canvas, které se nastavuje ještě před voláním první operace malování. Volání pak může vypadat nějak takto:

```
//init
…
MujObjekt.SetCanvas(gCanvas);
…
//malovani
…
```

```
MujObjekt.Draw();
```

...

Poslední zajímavou možností je, že se Canvas umístí do pozice SINGLETON (viz kniha Design Patterns v OOP), protože se používá vždy jen jedna „globální“ instance. Dohoda je taková, že aplikace ovládající hráčky „čte“ Canvas stále z této „globální“proměnné v posici SINGLETON, přičemž klient aplikace může dosadit do reference na tento objekt svůj vlastní Canvas.

2.16 Příklad na anonymitu klienta - autentizace klienta

Mnohdy bývá nutné řešit i tzv. bezpečnost informačních systémů.

Po aplikaci typu GSM Baniking (viz příklad na Monitoring) jsme následně řešili tzv. Internet Banking (tj. ovládání bankovních služeb standardním HTML browserem přes veřejnou síť internet). U aplikace typu Internet Banking je otázka bezpečnosti ještě palčivější: Na jedné straně internet je díky své otevřenosti velmi „nebezpečnou sítí“, na straně druhé bankovnictví obecně vyžaduje velmi vysokou bezpečnost.

U systémů vyžadujících bezpečnost nastává jeden problém, který souvisí s anonymitou klienta. Nazývá se autentizace (někdy je také česky uváděno jako autentikace).

Proces autentizace v informačním systému ověřuje, že subjekt vystupující vůči informačnímu systému je tím subjektem, za který se vydává. Většinou se jedná o ověření pomocí jména a hesla, což je sice nejčastější způsob, ale ukazuje se, že jednoduchá autentizace pomocí jména a hesla je mnohdy nedostatečná. Existují proto specializované prvky SW jako čipové karty, USB tokeny, externí tokeny apod., které proces autentizace podporují.

Pokud domyslíme do důsledku „oč jde“ v procesu autentizace, tak jsme doslova a do písmene v praxi narazili na problém anonymity klienta, nyní chápaného jako klienta celého systému. Pokud „personifikujeme“ systém, tak tento systém nikdy „neví“ (a ze samé podstaty věci ani nemůže vědět), kdo je na druhé straně jako klient. Svět je podle OOAP rozdělen fyzicky na dvě části: Uvnitř systému a vně systému. To, co je vně systému (klient) může přes interface systému poslat tomuto systému libovolnou zprávu, cokoliv, co je tento systém schopen přijmout. Klient například může systému poslat zprávu a ve vstupním parametru této zprávy říci „já jsem ten a ten“, ale to by však mohl říct každý! Uvnitř systému se zprávy vyhodnocují. Kvalitní proces

autentizace je takový, který při těchto předávání zpráv zabrání podvrhu, tj. to už by nemohl říct každý.

Pokud dojde k autentizaci, neznamená to narušení anonymity klienta a že poté „víme, kdo je na druhé straně“. Prostě pouze proběhl proces, kdy někdo anonymní na druhé straně vně systému poslal takové zprávy systému (například zadal správné jméno a heslo, použil správně externí token apod.), že po průběhu tohoto procesu je považován za autentizovaný subjekt.

Otázkou je, jakou podobu musí mít takový proces, který je schopen autentizace. To je však problém specializovaných prvků SW a zařízení.

2.17 Příklad na chybné chápání efektu zapouzdření

Velmi často se u začátečníků v OOP chybně chápe pojem zapouzdření a zaměňuje se s jakýmisi „utajením“, resp. „neposkytnutím nějaké informace“ ze strany objektu apod. Objekt je chápán spíše jako přísně utajená struktura s tajuplným vnitřek, která není příliš sdílná.

V OOP je pro efekt zapouzdření synonymem tato věta: „Jediná možnost, jak může klient použít objekt, je přes interface a nijak jinak“. Je pochopitelné, že tento interface musí poskytnout vše, co klient případně bude potřebovat. Jediné, co klient nemůže tušit, je to, jak tuto informaci vnitřně objekt zplodil a vytvořil.

V jedné firmě jsem možnost pracovat na projektu, kde bylo mimo jiné cílem zavést objektově orientované programování. Jeden z podřízených, říkejme mu třebas Laďa, za mnou přišel a sdělil mi: „Mám problém. Jura programuje jeden objekt, já ho mám používat a potřebuji identifikátor toho objektu, tedy ‘ídečko‘ . Ten objekt je zapouzdřen, to je v pořádku, ale Jura nezavedl žádnou operaci, ze které bych toto ‘ídečko‘ dostal. Já to ídečko fakt potřebuji a nemohu ho nikde dostat. Říkal jsem mu, ať ho teda vyvede přes nějakou operaci a dá mi ho jako návratový výstupní parametr, ale on to odmítl. Ale já to ídečko fakt potřebuji!“

V té chvíli jsem si připadal jako tatínek na pískovišti, kde se děti hádají o lopatičku a kyblíček. Zavolal jsem si Juru: „Juro, poslouchej, ty máš v objektu ídečko a nechceš ho dát Laďovi. Proč mu ho nechceš dát?“

„Protože to je jeho vnitřní věc, to idečko, a nemá ho co dostávat. Pokud bych mu ho dal, narušil bych zapouzdření.“

Ukázali jsme si přesně místo v programu, kde ho Laďa potřeboval. Přesto Jura odmítl ídečko vydat údajně z důvodu zapouzdření. Nakonec (podobně jako mezi dětmi na pískovišti) dostal příkazem ídečko vydat přes operaci (přes tzv. property).

3. Analytické vrstvy informačního systému

Tato kapitola patří k důležitým v celém výkladu o efektivním modelování informačních systémů pomocí UML. Ujasníme si, jaké jsou nosné myšlenky analytického modelování v UML.

3.1 Analytické vrstvy

Pro pochopení podstaty analytického modelování je velmi důležitý fakt, že ať už analytik modeluje libovolný informační systém, vždy vidí tento systém „stejným způsobem“ a tedy podle téže „šablony analytického myšlení“. Stejná a stále se opakující „šablona myšlení“ mu umožňuje popsat libovolný informační systém jednotným způsobem jednoduše, srozumitelně, jednoznačně a velmi přesně. Důležité je, že tento způsob popisu je z jedné strany velmi dobře srozumitelný jak „laikovi, uživateli“ (samozřejmě nikoliv přímo v notaci UML, ale zprostředkovaně jako vysvětlované modely), na straně druhé je velmi dobře srozumitelný designérovi a programátorovi. Tento náhled označme dále jako ANALYTICAL VIEW a budeme je chápat jako vzor (šablona).

Co se opravdu mění projekt od projektu, je obsah tohoto pohledu. Forma včetně způsobu uvažování je stejná. Nyní je třeba si vysvětlit, jak tato „jednoduchá šablona“ analytického uvažování vlastně vypadá.

První základní představa analytika o informačním systému spočívá v tom, že ať už je systém navržen následně v designu jakkoliv a programátorem napsán v jakémkoliv prostředí, ať už je program navržen strukturálně nebo objektově, případně ať je rozvržen do komponent nebo naopak napsán jako monolit, neboli ať už se jedná z hlediska designu o libovolný informační systému, tak vždy existuje v pohledu analytika nejvnitřnější analytická vrstva analytických tříd a výskytů z těchto tříd.

(Důležitá poznámka: Je třeba zdůraznit, že nyní nejsme na úrovni designu, ale na úrovni analytického modelování a pojem třída zde nemá povahu třídy v OOP jako například JAVA, DELPHI třídy, ale zde hovoříme o třídě analytického modelování).

Z hlediska popisu informace se při modelování analytik může pohybovat ve dvou možných rovinách:

- buď se v AM popisuje informace (pojem) v obecnosti a udává vlastnosti takto zavedených informací, tj. druhů informace
- anebo se v AM popisuje konkrétní výskyt informace.

V prvém případě se popisuje typ informace a v druhém případě se popisuje výskyt tohoto typu. Podobně se vyjadřuje nejenom analytik při modelování, ale každý, kdo se podílí na analytickém modelování. Tento přístup odpovídá normálnímu lidskému uvažování (viz kapitola pojednávající o objektově orientovaném přístupu).

Typ informace budeme nadále nazývat třídou AM, nebo analytickou třídou resp. třídou informace nebo typem informace. Tuto třídu nelze zaměňovat s pojmem třída v OOP. Té odpovídá třída v designu, tj. konkrétní třída v objektově orientovaném

jazyce. Třída v AM je pojem, se kterým se pracuje v IS, oproti tomu třída v OOP je prvkem kódu v daném jazyce (doslova `class` jako rezervované slovo).

Analytické třídy reprezentují evidované pojmy. V následném kroku návrhu designér tyto evidované pojmy nějak realizuje ve „svém“ prostředí více či méně objektově. Analytické třídy jsou nakonec realizovány jako skupiny softwarových prvků, kterými mohou být například tabulky, třídy v OOP, funkce, proměnné, soubory atd. Nemusí se vždy nutně jednat o objektově napsaný program.

Počet prvků ve vrstvě tříd (tj. počet analytických tříd, počet analytických typů) je neměnný. Vrstva tříd se proto chápe jako statická. Z toho důvodu se někdy model této vrstvy nazývá také jako statický model tříd.

Synonymem pro analytickou třídu jsou výrazy jako „evidovaný pojem“, „entita“, „evidovaná informace“ resp. jiné výrazy označující „co se vlastně eviduje“. Základním posláním analytických tříd je dávat „do vínku“ vlastnosti budoucím evidovaným výskytům těchto tříd.

Příklady tříd z bankovního systému: Osoba, Účet, Úvěr, Právnická osoba, Ručitel, atd. Tyto pojmy budou nakonec v evidenci systému v designu „nějak“ navrženy, například jako kombinace tabulek a objektů, nebo tabulek a funkcí apod.

Poznámka: Připomeňme, že ve fázi AM se nejedná o třídy „mimo systém“, ale „uvnitř systému“. Jinak řečeno nehovoříme nyní o třídách v modelování podniku (i když je zřejmé, že také v okolí existují třídy stejného názvu, jako Osoba, Účet apod., ale ty jsou předmětem modelování podniku). Tuto problematiku jsme již probrali v kapitolách o modelování podniku.

Analytik si představuje, že z takto zavedených analytických tříd, které stojí nějak uvnitř systému jako statický genotyp, vznikají v běhu programu (v run-time) výskyty z těchto tříd. Analytické výskyty jsou realizovány v designu nějakou skupinou výskytů prvků programu, například jako objekty v programu napsaném pomocí OOP, nebo pomocí volání funkcí s konkrétní daty s hodnotami, skupina proměnných v paměti apod.

V představě analytika tak vzniká „druhá“ vrstva a to vrstva výskytů z analytických tříd. Počet výskytů je již v čase (v běhu programu) proměnný, tedy tato vrstva je dynamická. Výskyt z třídy AM budeme nazývat také jako instance informace nebo výskyt informace, někdy také jako objekt, ale opět tento pojem nelze zaměňovat s naprogramovaným objektem v OOP. Z hlediska předešlého výkladu můžeme typ informace neboli třídu v AM považovat za obdobu zavedeného pojmu a výskyt za evidované výskyty těchto pojmů.

Počet prvků vrstvy výskytů není stálý a je v běhu programu proměnný. Výskyty se chovají tak, že v systému v běhu programu vznikají a to s vlastnostmi, které získaly ze tříd ze statické vrstvy. Následně ve svém životě interagují mezi sebou tak, že se navzájem používají (např. při výpočtech apod.), případně zanikají jako již nepotřebné. Tato vrstva výskytů je charakteristická dynamickým chováním výskytů.

Příklady: Evidovaná faktura číslo 1010, Osoba s rodným číslem 5512152367, Účet číslo 2343453567 apod.

Vztah mezi vrstvou tříd a vrstvou výskytů je již jednou zmíněný vztah „meta“, tj. vztah mezi typem a jejím výskytem. Znamená to, že obě tyto vrstvy je možné chápat jako dvě části jednoho celku.Vrstva má tedy dvě úrovně „meta“: Jedna úroveň se týká typů informace (tj. tříd) a druhá výskytů informace.

3.2 Příklad na analytické třídy a analytické instance

Část rozhovoru analytika se zákazníkem - odběratelem informačního systému v bance může vypadat takto:

Zákazník: *„Budou se evidovat běžné účty. Každý účet má tyto vlastnosti...(následuje výčet vlastností)“*

Analytik: *„Kolik tak řádově předpokládáte, že by tak asi mělo těchto účtů v evidenci být, samozřejmě plus minus...“*

Zákazník: *„Řádově tak okolo 800 000, možná víc, možná méně...“*

Všimněme si, že se jedná se o klasický rozhovor na úrovni abstrakce analytického modelování AM. Nehovoří se o tabulkách resp. konkrétních prvcích programu, ale oba účastníci rozhovoru se vyjadřují pouze v pojmech tak, jak vyžaduje tato abstraktní úroveň. V první části rozhovoru se zákazník vyjádřil o vlastnostech účtů jako takových a hovořil tedy o třídě běžných účtů jako o vlastnostech druhu evidované informace, ze kterých budou pocházet instance. Ve druhé části rozhovoru, kde se „počítaly“ instance, se hovořilo o výskytech této informace, tj. o počtu výskytů ze třídy běžných účtů.

Je možné, že daný program bude realizován čistě strukturálně pomocí dat a funkcí, například jako funkce obsahující SQL s relační databází. Znamená to, že analytické třídy a analytické výskyty (o kterých byla v rozhovoru řeč) budou nakonec namapovány na tabulky a záznamy v tabulkách, dále na funkce a jejich volání.

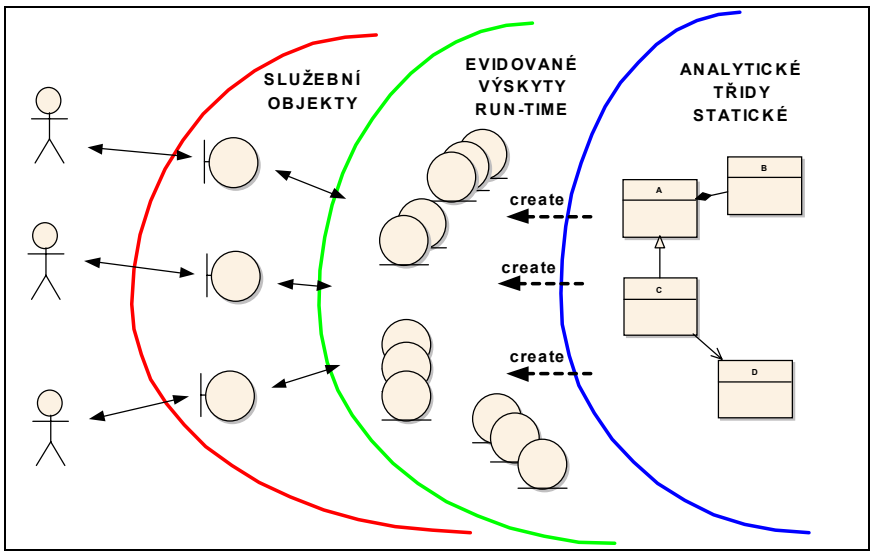
3.3 Vzor ANALYTICAL VIEW

Z pohledu analytika existuje ještě jedna vrstva, kterou nazveme „služební vrstva“, který vzniká nutností existence prvků, které služebně zprovozňují systém (obrazovky, prvky GUI obecně, parsery apod.). V této vrstvě také existují nějaké třídy a výskyty, avšak tyto nemají skutečnou analytickou povahu jako měly třídy a výskyty v předchozí vrstvě. Tyto třídy a výskyty analytik nepopisuje přímo jako analytické entity, ale určuje jejich vlastnosti nepřímo pomocí popisu interakcí mezi okolím systému na straně jedné a výskytů z analytické „dynamické vrstvy výskytů“ na straně druhé. Analytik takto popisuje analytický obsah této interakce okolí versus výskyty informace nikoliv konkrétně jak bude realizována, ale velmi přesně modeluje skutečný analytický obsah této interakce. Tím dává designérovi zadání pro návrh a použití služebních objektů v designu.

Příkladem takového přesného určení interakce ve služební vrstvě je například tento popis: *„Obsluze se zobrazí seznam fyzických osob (rodné číslo, jméno, příjmení)…“* Všimněme si, že je sice analyticky popsáno přesně, co se děje vůči okolí, ale nejedná se o konkrétní implementační podrobnosti designu. Uvedené zobrazení seznamu bude realizováno nějakým konkrétním služebním objektem, například pomocí `ListView` apod.

Obecně služebními objekty mohou být například prvky formuláře, objekty pro export a import informací, objekty pro přehledy a výpisy, parsery, objekty pro přístup do databáze aj.

Celkovou představu o vrstvách aplikace v analytickém modelování, tj. ANALYTICAL VIEW, znázorňuje následující obrázek.



obrázek 15 ANALYTICAL VIEW

Tento obrázek je třeba chápat jako vzor: Analytik vždy takto systém vidí a vyplňuje účastníky vzoru konkrétním popisem prvků (třídy, výskyty, chování výskytů).

Systém je na této úrovni abstrakce viděn tak, že existuje nejvnitřnější analytická vrstva, tzv. vrstva analytických tříd. Na obrázku je vyznačena úplně vpravo modře. Jedná se o vrstvu pro „rodiště výskytů“, jinak řečeno pro jejich genotyp. Analytik tuto vrstvu popisuje pomocí modelu tříd v UML. Z této vrstvy budou vznikat výskyty evidovaných informací v systému a tato vrstva jim dává vlastnosti jako množina jejich typů.

Je na designérovi, aby tuto vrstvu tříd nějak realizoval ve svém prostředí. V jeho prostředí bude tato nejvnitřnější vrstva analytických tříd konkrétně namapována na množinu nějakých typů daného prostředí.

Uvažme například klasický hybridní systém, tj. systém se silnou relační databází a silným OOP jazykem. Nejčastějšími kombinacemi hybridních systémů jsou „JAVA plus ORACLE“ nebo „DOT NET plus MS SQL“, případně jiné kombinace. V tom případě se nejvnitřnější vrstva analytických tříd mapuje na dvě množiny typů: Jednak do relační databáze na tabulky a za druhé na konkrétní třídy daného objektového jazyka. Pokud by se však systém realizoval například v Pascalu 7.0 pomocí objektů a souborů, tak by se mapovalo na množiny jednak typů tříd v Pascalu plus na množinu typů pro soubory (například `object` a `record` apod.).

Jak vidět na obrázku, další vrstvou je dynamická vrstva výskytů informace, na obrázku prostřední zelená vrstva. Tato vrstva vzniká tak, že v běhu programu vznikají z analytických tříd analytické výskyty informace, případně i zanikají. V běhu programu mezi sebou tyto výskyty nějak interagují, tj. nějak se chovají. Analytik

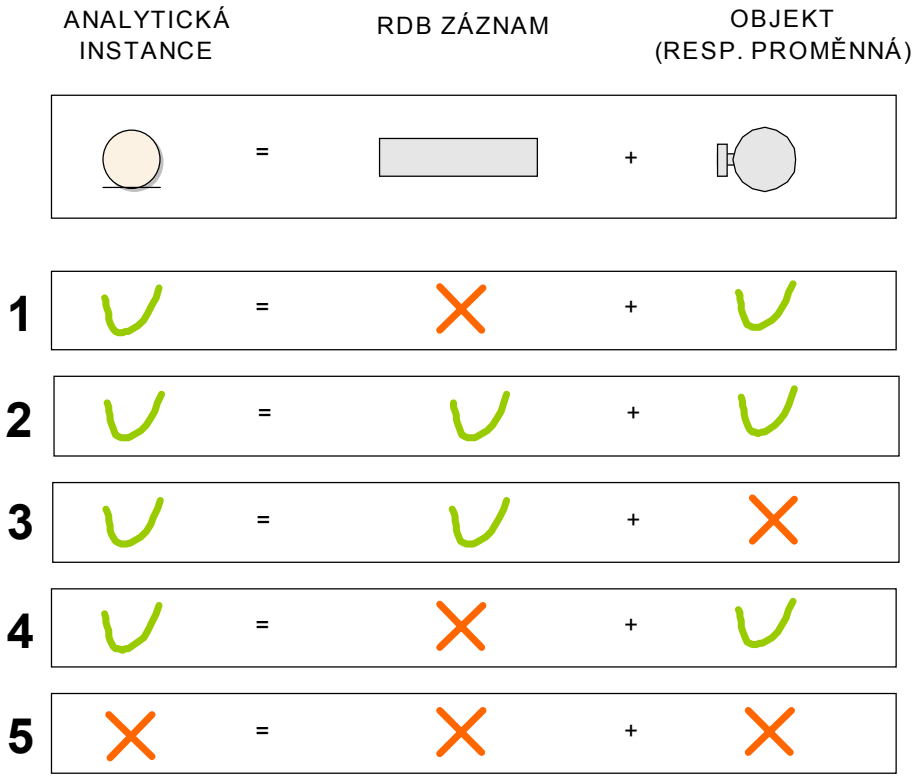
popisuje toto chování opět pomocí modelů UML. Díky bohatému jazyku UML má na výběr hned z několika možných typů modelů, které jsou schopné popisovat chování výskytů (tzv. dynamické modely). Ukazuje se, že pro evidenční systémy má mezi nimi výsadní postavení USE CASE MODEL spolu se zavedenými slovními scénáři (bude podrobně rozebráno dále).

Například v uvedeném hybridním systému (JAVA plus ORACLE nebo DOT NET plus MS SQL) jsou analytické výskyty namapovány na kombinace dvou „fyzických“ výskytů: Na řádek (záznam) v tabulce plus alokovaný objekt v paměti. Znamená to, že „jeden analytický výskyt“ (jak jej vidí analytik) je mapován na dvojici výskytů (jak je vidí designér). Není bez zajímavosti, že právě zde, v této představě, dochází k nejčastější chybě v chápání, jak má vlastně vypadat takovýto „hybridní výskyt řádek plus objekt“. Podstata této chyby spočívá v tom, že se použije velmi zjednodušená představa mapování analytických výskytů do OOP objektů a relační databáze.

Tuto chybu si vysvětlíme tímto příkladem: „V aplikaci se má vyskytovat asi jeden milion účtů. To mají být všechny instance účtů alokovány v paměti?“ Tato představa vychází z mylné představy, že existuje jedno jediné primitivní mapování mezi analytickými instancemi a objekty v paměti a to je mapování „jedna ku jedné“. Pro každou logickou instanci, tj. pro každou instanci v analytickém modelování, by v tomto případě měla být alokována jedna instance v OOP. To je mnohdy technicky nemožné. Existuje však několik dalších možných mapování, takových, že analytické instance versus objekty v paměti nejsou jedna ku jedné. Mimochodem, tento problém není nic nového, například z úplně stejného důvodu vznikl vzor FLYWEIGHT (viz kniha Design Patterns v OOP).

3.4 ANALYTICAL VIEW a hybridní systémy

Je dobré si uvědomit, že pro hybridní systémy platí, že zatímco analytik „vidí“ v systému jednu evidovanou analytickou instanci, tak designér v hybridním systému vidí dvojici instancí: „Něco je v paměti“ a „něco je v databázi“. Teprve až kombinace těchto dvou fyzických instancí dává dohromady to, co analytik vidí jako jednu analytickou instanci. Navíc obě instance v designu procházejí stavy, kdy dané instance z dvojice instancí buď existují anebo neexistují. To je v následujícím obrázku znázorněno „zelenou fajfkou“ jako ANO a „červeným křížkem“ jako NE:



obrázek 16 Pohled analytika a designéra na instance informace v hybridním systému

Stav označený jako 1 odpovídá situaci v hybridním systému, kdy v paměti (např. v proměnných, v objektech apod.) je informace již vyplněna, avšak ještě není uložena v databázi. Pokud by v této chvíli systém tzv. „spadl“ (např. někdo jej vypnul), tak se informace ze systému ztratí.

Stav označený jako 2 odpovídá situaci, kdy jak v paměti tak v databázi jsou v rámci dané dvojice instancí informace se stejnou hodnotou, tj. instance jsou v „synchronu“, obě žijí, jak instance v paměti, tak v databázi. Právě zde se nachází největší problém s chápáním hybridních „RDB+OOP“ aplikací. Tento stav není povinně vyžadován pro úplně všechny instance najednou, ale například pouze pro několik málo instancí. Je pouze na designérovi, zda tohoto stavu využije anebo nikoliv a pokud ano, tak do jaké míry. Některá prostředí umějí velmi rozumně „handlovat“ se zmíněnými stavy v tom smyslu, že v paměti jsou hodnoty z databáze pouze pro instance, které jsou takzvaně otevřeny pro použití (tzv. cache systémy). U více-uživatelských systémů, například s vícero uživateli na síti, není problematika těchto systémů až tak jednoduchá, jak na první pohled vypadá.

Stav označený jako 3 může být buď stavem, kdy se s danou instancí v dané části programu nepracuje, tj. v daném popisu dynamiky chování v analytickém scénáři instance „spí“ a nic se s ní neděje, resp. nikdo se systémem nepracuje apod., anebo

se jedná o zvláštní způsob mapování, kdy z technologických důvodů nelze použít mapování do OOP a proměnných do paměti a musí se využít tzv. STORED PROCEDURA uložená přímo v RDB. V tom případě „v paměti“ nic nežije a vše proběhne přímo nad záznamy v RDB. I to je však z hlediska analytického modelování relevantní stav analytické instance v systému, zde označený jako stav 3.

Stav označený jako 4 vypadá stejně, jako stav 1, avšak chceme zde poukázat na možnost stavu, kdy v paměti ještě nějaké údaje jsou, ale v databázi jsou již vymazány (na rozdíl od stavu 1, kde jsme hovořili v paměti už jsou a v databázi ještě nejsou uloženy). Až instance zmizí i z paměti, přestane žít instance v analytickém pojetí.

Stav 5 odpovídá právě tomu, kdy neexistují ani údaje v paměti a ani v relační databázi. Jedná se o stav, kdy buď instance ještě vůbec nebyla zadávána anebo byla vymazána jak z databáze, tak z paměti.

Všimněme si, že analytik vidí systém mnohem jednodušeji, protože je oproštěn od specifického prostředí hybridního systému „RDB plus OOP“: V levém sloupci vidí pouze stav 5 jako „NE“, ostatní chápe jako „ANO“. Technologické vnitřní stavy vztahu instancí mezi pamětí a databází jej nezajímají.

Pozn.: Je třeba ještě poznamenat, že existuje zvláštní případ mapování do hybridního systému, kdy se používá zejména a pouze stav 3 a pouze chvilkově něco proběhne přes paměť. Toto mapování do hybridního systému je hodně specifické. Jedná se o systémy, u kterých z technologických důvodů nelze používat klasické objekty jako alokované struktury v paměti se svou vnitřní pamětí s hodnotami v atributech zejména z důvodu velkého počtu současně připojených uživatelů. Tomuto mapování se říká „stateless programming“ a využívá zvláštního mapování, kdy stav, tj. informace v instancích jsou namapovány pouze do databáze, přičemž hodnoty v paměti jako takové nežijí. Ke stavu se zelenou fajfkou u objektů de facto nedojde, všechny proměnné pouze „proplynou“ co nejrychleji „do a z“ RDB.

Vraťme se k obrázku viz *obrázek 15 ANALYTICAL VIEW*, kde je patrná ještě nejvrchnější služební vrstva, která obsahuje služební objekty daného prostředí (viz červená vrstva). Tato vrstva pracuje s výskyty ze zelené vrstvy. Dá se říci, že služební objekty z červené vrstvy „obsluhují“ instance ze zelené vrstvy. Služební objekty například zobrazují údaje z analytických instancí na obrazovku, nebo obráceně přejímají údaje od obsluhy a předávají je instancím z analytické vrstvy, případně importují údaje přes nějaké rozhraní apod. O služební objekty se analytik nestará v tom smyslu, že by je konkrétně specifikoval. Pouze podrobně popíše chování „co se děje s výskyty“, tj. jak se s nimi pracuje a tím designérovi předává zadání pro funkcionalitu těchto služebních objektů.

Uvedený *obrázek 15 ANALYTICAL VIEW* lze chápat jako vzor (ANALYTICAL PATTERN) v tom smyslu, že každý informační systém podléhá tomuto analytickému pohledu a lze tedy takto na něj vždy nazírat. Zmíněný analytický náhled označený jako ANALYTICAL VIEW je platný vždy pro každý informační systém. Jedná se tedy

o jakousi „šablonu“ platnou pro všechny informační systémy. Znázorněné vrstvy existují v tomto pohledu vždy, pouze je třeba specifikovat jejich obsah.

Vyplývá z toho, že hlavním úkolem analytika je dobře a správně vyjádřit a vyplnit konkrétně tento pohled na uvedeném obrázku viz *obrázek 15 ANALYTICAL VIEW* pro konkrétně řešený systém. Analytik v tomto vzoru „vyplní“ šablonu vrstev, čímž předává zadání pro designéra. Tento analytický pohled se stává zadáním a následně se realizuje do konkrétních softwarových prvků daného prostředí ve fázi designu.

Pro vyplnění zmíněné šablony, tj. pro tvorbu konkrétního popisu vrstev na obrázku viz *obrázek 15 ANALYTICAL VIEW* pro konkrétní informační systém analytik použije modely UML, které zde v této knize postupně probereme.

Není překvapivé, že mezi těmito modely je velmi důležité použití modelu typu CLASS MODEL jako analytický model tříd. Tento analytický model tříd popisuje nejvnitřnější vrstvu (viz modrá vrstva na obrázku), tj. genotyp aplikace neboli rodiště pro evidované výskyty informací v aplikaci.

Z toho důvodu začneme výklad o modelování kapitolou Analytický model tříd.

4. Analytický model tříd - CLASS MODEL ve fázi analytického modelování AM

Jedním z hlavních cílů analytika ve fázi analytického modelování je nalézt odpovídající model tříd, tj. CLASS MODEL AM. Tento analytický model tříd se stává výchozím pro mapování tříd do designu a na základě něj se vytváří odpovídající struktury programu, např. návrh tabulek v relační databázi (dále také RDB) a návrh tříd v OOP. Po zhotovení programu tyto struktury v designu musí přesně odpovídat takto navrženým typům informací v analytickém modelu tříd.

Připomeňme analytický náhled na informační systém z předešlé kapitoly: Uvnitř systému existují naprogramované třídy, ze kterých se rodí instance a ty během svého života realizují chod informačního systému.

4.1 Zavedení třídy v analytickém modelování a problém určení meta-úrovně pojmů

Analytik při tvorbě analytického modelu tříd informačního systému zavádí jednotlivé analytické třídy. Vychází se přitom z konzultací s budoucím uživatelem resp. konzultantem anebo z vlastních znalostí analytika o systému. Základními otázkami pro vznik analytické třídy jsou otázky „co se bude evidovat“, „jaký pojem bude evidován“, „co se eviduje“, „z čeho budou evidované výskyty“ apod.

Zavedením třídy analytik vlastně předává designérovi zadání v tomto duchu:

„Až bude systém hotov, rád bych, abych při pohledu na něj viděl chování výskytů z této třídy nějak podle tebe navržené. Bylo by chybou, kdybych tam tyto pojmy neviděl anebo kdybych u nich viděl jiné vlastnosti, než jaké ti v modelu tříd zadám.“

Již zde se však skrývá určité úskalí. Při rozhovoru s konzultantem resp. při analýze problémové domény se začnou vyskytovat určité pojmy určené k evidenci, tj. pojmy, které se stávají kandidáty na evidované analytické třídy. Ale zde již může dojít k velmi časté chybě, která spočívá v záměně instance a třídy. Na první pohled totiž nemusí být vůbec jasné, zda daný pojem, o kterém je v konzultacích řeč, spadá do oblasti evidovaných instancí anebo do oblasti evidovaných tříd.

Při vymezení nové třídy musí být analytik obezřetný, zda zkoumaný pojem skutečně patří do oblasti tříd anebo do oblasti instancí. Mnohdy totiž dojde k záměně, kdy to, co bylo považováno za třídu, je nakonec navrženo jako instance nějaké jiné třídy. Tuto chybu budeme nazývat *„chybné určení meta-úrovně“*. Dále jsou uvedeny klasické příklady na chybné určení meta-úrovně.

4.2 Příklad na určování meta-úrovně – flexibilní dotazník

V jedné ostravské firmě řešili jednu zajímavou aplikaci, nazvěme ji *Flexibilní dotazník.* Podstata této aplikace byla následující: Pokud bychom si koupili takovýto *Flexibilní dotazník*, tak hned po spuštění bychom vstoupili přes menu do části aplikace zvané „konfigurace dotazníku“. V této části aplikace bychom si sami zadali

své otázky, k nim správné odpovědi, možnosti odpovědí apod. Po této konfiguraci se vytvoří nový dotazník, přesně takový, jaký potřebujeme. Takovýchto dotazníků si v aplikaci můžeme vytvořit kolik chceme.

Nějaký námi vytvořený dotazník budou vyplňovat různé evidované osoby, k nim se budou evidovat výsledky. Můžeme si představit, že takovouto aplikaci si koupí například České aerolinie (ČSA) a pomocí této aplikace si vytvoří zkoušky pro piloty. Stejnou aplikaci si koupí například nějaká vysoká škola a vytvoří si pomocí ní například přijímací zkoušky. Armáda si vytvoří zkušební testy pro profesionální vojáky, autoškola zkušební testy pro řidiče atd.

Otázka zní jednoduše: Bude se v této aplikaci vyskytovat třída „Zkouška pilotů ČSA“? Nebo třída „Přijímací zkouška na VŠ XY“? Nebo „Zkušební test pro řidiče“? Nikoliv, pojmy v uvozovkách spadají až do oblasti instancí a reprezentují konkrétní hodnoty názvů dotazníků.

Jaké tedy budou třídy? Odpovězme pouze asi tak přibližně, modelovat ještě neumíme: Kandidáty na třídy jsou například pojmy jako „dotazník“, „otázka“, „odpověď“ apod. Z nich budou vznikat instance s konkrétními hodnotami. Kromě toho se budou evidovat lidé, kteří odpovídali na otázky.

Všimněme si, že chybné určení meta-úrovně souvisí velmi úzce s možnostmi „zobecnění“ do úrovně meta, tj. do typu. Podstata flexibilního dotazníku vyžaduje, aby určité hodnoty vznikaly až v run-time a proto nemohou být třídami, protože se jedná se o konkrétní hodnoty vzniklé až v běhu programu.

Tyto úvahy navíc ještě pro designéra znamenají, že prvky v obrazovkách musí také navrhovat jako flexibilně a dynamicky tvořené v run-time, nikoliv „natvrdo v kódu“. Určitě v kódu nenajdeme obdobu takovéto „tvrdé“ konstrukce (symbolicky zapsáno):

```
MyForm.Caption = 'Dotazník pro ČSA' ;
```

ale spíše něco v tomto duchu nebo obdobně:

```
MyForm.Caption = MyDotaznik.Nazev() ;
```

4.3 Chyba úhybného manévru analytika do metasystému

V souvislosti s určením „správné“ meta-úrovně je třeba místě upozornit na jednu velmi častou záludnou chybu. V některých případech se zobecňování do meta-úrovně přežene a vzniknou něco jako „meta-super-systémy“, které umějí všechno, ale ve skutečnosti vlastně neumějí nic. Tato chyba vzniká většinou při neznalosti problémové domény a při velmi slabé analýze. Z důvodu neznalosti „jak to má vlastně být“ se začne navrhovat a programovat „něco“, co má být nějakým způsobem univerzální pro všechny možné případy, které by díky neznalosti mohly nastat a o kterých nevíme. Výsledkem je systém, který se nakonec „doprogramovává“ v jakémsi podivném a velmi nepřehledném prostředí metasystému. Vznikne systém, který řeší nějaké obecné problémy a nikoliv samotnou aplikaci. Například namísto nějaké požadované bankovní agendy vznikne klasické repository objektů, nadstavba nad databází apod. Navíc pokud je systém navržen s chybami, některé konstrukce nakonec stejně nelze navrhnout a musí se do řešení pracně zasahovat.

Samozřejmě nemám na mysli tu situaci, kdy je takovýto systém přímo v zadání ve smyslu předešlého příkladu *Flexibilní dotazník*. Pokud je zadání přímo v duchu „udělejte takovýto dotazník s takovými vlastnostmi“, tak se jedná o konkrétní zadání konkrétní aplikace. Pod touto chybou mám na mysli jakýsi „úhybný manévr“, kdy se žádá dodat určitá aplikace s konkrétními funkcionalitami, což je zadání, a autoři systému však z důvodu neznalosti podrobností těchto funkcionalit „sklouznou“ do metasystému. Začnou řešit úplně jinou agendu s myšlenkou „ono to konkrétní, co ještě nevíme, se tam nakonec bude moci nějak dosadit“. Chyba nespočívá v samotném řešení metasystému, ale v tom, že zadání zní jinak, než je konečné řešení.

Je pochopitelné, že při řešení dané konkrétní agendy s konkrétním zadáním je vhodné, aby zůstala tzv. „otevřená vrátka“ pro dobré, snadné a přehledné rozšíření systému, tj. aby řešení aplikace bylo flexibilní. Ukazuje se, že flexibilní a dobře rozšiřitelný systém vznikne jako kombinace dvou faktorů, které musí vývojáři na všech úrovních velmi dobře ovládat: Musí se použít opravdu dobrý a kvalitní analytický návrh, který správně používá všechny možné vazby modelu tříd (včetně dědičnosti) spolu se správným použitím návrhových vzorů v OOP. V tom případě vznikne velmi dobrý aplikace „šitá na míru“ se současně otevřenými vrátky pro další extenzi.

Poznámka: Jenom na okraj je třeba podotknout, že uvedené chyby „sklouznutí do metasystému“ se nejčastěji dopouštějí ti vývojáři, kteří jsou příliš dlouho zvyklí

pracovat ve strukturálním prostředí a pracují bez znalosti objektových principů a principů analytického modelování. Jejich touhou je tvorba univerzálních systémů, protože jejich „ne-objektové návrhy“ nejsou vůbec flexibilní.

4.4 Třída multiinstanční a třída s konkrétním počtem instancí

Když analytik navrhuje danou analytickou třídu, měl by udat, zda je třída tzv. multiinstanční anebo zda má omezený počet instancí. Ve většině případů je třída tzv. multiinstanční, což znamená, že z daného druhu typu informace lze vytvořit obecně N výskytů informace. Pouze výjimečně existují takové typy informací, které mají omezený počet výskytů, většinou pouze jeden výskyt. Pro tyto situace se použije vzor zvaný SINGLETON (viz e-kniha Design Patterns v OOP). Jako příklad jedno-instanční informace uveďme případ, kdy se v programu typu desktop, například v kancelářském softwaru, zadávají detaily uživatele tohoto softwaru.

Protože většinou je třída multiinstančí, je vhodné, aby existovala ve firmě dohoda, že pokud není uvedeno jinak, třída informace se chápe automaticky jako multiinstanční.

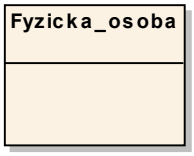
4.5 Omezená syntaxe modelu tříd analytického modelování

Analytik má pro tvorbu analytického modelu tříd díky syntaxi UML k dispozici několik málo pravidel, které si nyní vyjmenujeme. Díky této skutečnosti se modely tříd AM stávají přehlednými, jednoznačnými a syntakticky přesnými. V dalších kapitolách následuje výčet pravidel, která vedou k efektivnímu modelování.

Z důvodu přehlednosti a logické návaznosti vysvětlujeme jednotlivé prvky syntaxe v určitém pořadí tak, aby byly co nejvíce přístupné efektivnímu analytickému myšlení a tedy aby byly maximálně „schůdné“. K tomuto pořadí mne vede dlouholetá zkušenost konzultanta. Nejprve si tedy vysvětlíme, jak je třeba z pohledu analytika chápat prvky modelu tříd, následně si na konci kapitoly probranou látku uvedeme do celkového souladu s přesnou syntaxí modelu CLASS MODEL v UML.

4.6 Zavedení třídy v analytickém modelu tříd

Základní syntaxí modelu tříd je samotné zavedení pojmu tj. třídy v AM. Analytik tímto definuje hranice pojmu tj. typu informace. Základním posláním třídy jako typu informace je zavést předpis (neboli vlastnosti) pro budoucí instance z této třídy, tj. výskyty informace z této třídy. V daném CASE nástroji se v diagramu zavede nová třída informace s odpovídajícím názvem, například takto:



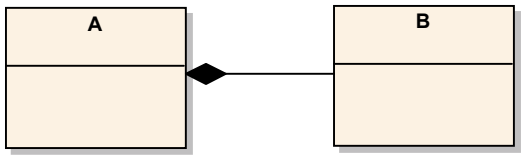
obrázek 17 Zavedení třídy Fyzická osoba v diagramu

Uveďme nyní několik důležitých pravidel:

- Jedná se o třídu ve smyslu abstrakce jako typ informace. Analytik požaduje, aby na základě této třídy vznikly mapováním do designu (nějaké) struktury programu nesoucí možnosti evidované informace (v tomto případě Fyzické osoby). Mapování do designu může být např. na tabulky v relační databázi a současně na třídy v OOP (viz předešlá kapitola). Tyto struktury v programu budou mít po naprogramování přesně takové vlastnosti, jak je v tomto analytickém modelu určil analytik.
- Pokud není řečeno jinak, třída je multiinstanční. Uvnitř systému tedy budou existovat evidované výskyty informace (v tomto případě fyzické osoby) jako konkrétní výskyty informací s vlastnostmi, které budou definovány v této třídě.
- Třída vymezuje hranice pojmu. Kdo použije tuto třídu například ve svém dalším modelu (vnější pohled), použije ji se vším, co s ní bude spojeno vnitřně.
- Zavedená třída se bude mapovat na odpovídající struktury programu v designu podle pravidel mapování.

4.7 Kompozice

Jako první vztah mezi třídami si probereme vztah zvaný kompozice. Kompozice patří k nejjednodušším vztahům v modelu tříd. Značí se spojnicí s černým kosočtvercem takto:



obrázek 18 Znázorněný vztah kompozice

Kompozice vyjadřuje vztah mezi instancemi informace, které ze třídy vzniknou, a chápe se jako vztah mezi instancemi „celek versus část“. Znamená to, že kompozice udává, v jakém vztahu budou vůči sobě instance ze třídy A a instance ze třídy B, až vzniknou. Současně určuje, že tento vztah mezi instancemi bude „celek versus část“. Černý kosočtverec označuje tu instanci, kterou chápeme jako celek ve vztahu. Jinak řečeno instance informace ze třídy A bude v sobě obsahovat instanci informace (resp. několik instancí) ze třídy B jako svou část (jako kompozit).

Toto obsažení celek versus část je natolik silné, že výskyt informace A ovládá veškerý život výskytu informace B jako jeho nadřízený prvek a to vždy a za každých okolností. Výskyt z B bude doslova a do písmene žít pouze jako část výskytu z A a nikdy jinak. Zrod a zánik výskytu B je pouze v kontextu výskytu z A a nikdy jinak.

Z toho plyne, že zrod instance ze třídy B je předurčen tím, že se musí narodit v kontextu instance ze třídy A. Přesně totéž platí i naopak pro vymazání: Existuje-li například scénář vymazání evidované instance ze třídy A ze systému (byla například omylem zadána), potom instance ze třídy A s sebou bere do záhrobí vždy i svou instanci (resp. instance) ze třídy B a to vždy bez výjimky. Výskyt ze třídy B jako část instance ze třídy A se rodí, žije a zaniká vždy u svého majitele, tj. v kontextu své instance ze třídy A a z toho nejsou výjimky. Instance ze třídy B nemůže nikdy žít „sama o sobě“ bez majitelství nějaké instance ze třídy A.

Příklad: Faktura obsahuje řádky faktury, tj. v našem vztahu na straně třídy A je Faktura, na straně B je třída Řádek faktury. Každá instance ze třídy Řádek faktury vzniká a zaniká v kontextu dané instance (majitele) ze třídy Faktury a nikdy jinak. Evidovaný řádek nikdy nevznikne sám o sobě a taky nikdy svou fakturu neopustí, nikam neputuje. Nemá smysl hovořit o evidovaném řádku faktury bez faktury, tj. řádku, který by žil sám o sobě.

Zapamatujme si, že základní vlastností kompozice je to, že se jedná o budoucí vztah mezi instancemi, z nichž jedna je celek a druhá část, což bude platit bez výjimky a vždy.

4.8 Multiplicita

U kompozice lze udat ještě další údaje spojené s tímto vztahem resp. s konci této interakce. Prvním z těchto údajů je tzv. multiplicita (označovaná také jako násobnost), která se udává na konci kompozice na straně u třídy B. Multiplicita udává, kolik je možných výskytů ze třídy B vůči výskytu ze třídy A. Vícenásobná multiplicita jako N výskytů se označuje hvězdičkou. Možné hodnoty této multiplicity jsou:

- 0..1 je povolena buď žádná instance nebo jedna instance ze třídy B v majitelství instance ze třídy A
- 1 je povolena právě jedna instance ze třídy B v majitelství instance ze třídy A
- * je povoleno N instancí ze třídy B v majitelství instance ze třídy A
- 0.. * je povolena žádná až N instancí ze třídy B v majitelství instance ze třídy A
- 1..* je povolena jedna až N instancí ze třídy B v majitelství instance ze třídy A
- a jiné možné kombinace.

Všude, kde se vyskytuje označení pomocí hvězdičky *, tj. N výskytů, se na straně instancí ze třídy B chápou tyto instance jako seznam instancí ze třídy B. Tento seznam instancí ze třídy B je při kompozici obsažen v instanci ze třídy A. Každá z instancí ze třídy B v tomto seznamu instancí je ve svém životě řízena instancí ze třídy A.

Příklad: V předešlém příkladu s Fakturou bychom konec vztahu na straně u třídy Řádek označili multiplicitou *.

Na údaje, které se týkají multiplicity *, musí být analytik velmi obezřetný, protože mohou vést k zásadním omylům. Vraťme s k příkladu z modelování podniku BM s fakturou, kdy od konzultanta zazněla věta „došlá faktura má alespoň jeden řádek“. To by mohlo vést k mylné domněnce, že evidované instance ze třídy Faktura budou mít vztah k instancím ze třídy Řádek faktury 1...*. Avšak to v žádném případě není vůbec přesné, protože ve stavu „otevřená k editaci“ může mít faktura také „žádný řádek“, tj. v tomto stavu je multiplicita 0..* řádků. Oproti tomu ve stavu „uzavřena“ má

multiplicitu 1..* řádků. Vidíme, že samotný rozsah (zda 0..* nebo 1..*) záleží na stavu dané uvažované instance. Z toho důvodu doporučuji zavést raději dohodu a udávat raději pouze obecnější multiplicitu ku N pouze hvězdičkou a poté specifikovat v jiných modelech další podrobnosti.

4.9 Jednosměrné a obousměrné vztahy

Pokud se vrátíme ke kapitole, která vysvětlovala objektově orientované paradigma a tím také ukázala, jak správně chápat prvky v modelech, tak nás nepřekvapí, že také v modelu tříd jsou všechny vztahy mezi třídami vždy chápány jako jednostranné. Pokud je vztah mezi třídami oboustranný, tak se jedná o dva jednostranné vztahy, i když v syntaxi UML pro zhutnění notace je oboustranný vztah znázorněn v diagramu pouze pomocí jediné spojnice. Avšak vždy, i když vidíme v UML pouze jednu spojnici jako vztah mezi instancemi, tak se jedná o vztah z jedné strany od jedné instance vůči druhé instanci a případně i naopak. Jinak řečeno případ obousměrného vztahu je chápán jako dva jednosměrné vztahy, i když v UML se v CLASS MODELU znázorňuje jedinou spojnicí.

Analytické modelování v modelu tříd zásadně používá důsledně toto rozdělení interakce na dva směry. Je třeba zdůraznit, že nyní ve výkladu o kompozici zatím nehovoříme o obráceném vztahu od instance ze třídy B k instanci ze třídy A, ale pouze o vztahu od instance ze třídy A k instanci (instancím) ze třídy B. K obrácenému vztahu dojdeme až dalším výkladem. Nyní tedy uvažujeme tu situaci, kdy někdo používá instanci ze třídy A a používá její vnější pohled. Instance ze třídy A má ve svém vnitřním pohledu (ve své implementaci) instanci resp. N instancí ze třídy B a to je v této fázi výkladu vše.

4.10 Role

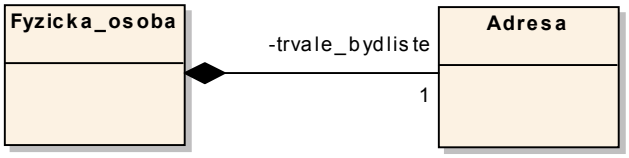
Dalším údajem, který by se měl doplnit u vztahu kompozice, je tzv. role třídy B vůči třídě A. Udává se, v jaké roli vidí instance ze třídy A jako majitel instanci ze třídy B. Je dobré si nyní uvědomit, že jedna a tatáž třída může být použita v různých interakcích a tedy může hrát různou roli. Role se značí v diagramu názvem této role na konci vztahu u třídy B. Role označuje význam instance resp. instancí z třídy B vůči A. Prakticky nejefektivnějším a nejčastěji používaným postupem je zavést jako roli přímo název této instance ze třídy B resp. název seznamu těchto instancí v případě N výskytů z B. Název této instance (resp. seznamu instancí) logicky

vystihuje i její roli. Význam role na konci vztahu kompozice nejlépe vysvětlíme na příkladu s evidencí bydliště u fyzické osoby:

Jako analytici zavedeme třídu Adresa, tato třída vytváří instance obsahující Ulici, Město a PSČ. V informačním systému chceme ke každé evidované fyzické osobě přiřadit evidovanou adresu trvalého bydliště jako kompozici ku jedné.

Poznámka: Je třeba upozornit, že existuje několik možných řešení, tj. několik možných modelů, ale protože vysvětlujeme vztah kompozice, zvolíme „úmyslně“ kompozici, i když netvrdíme, že se jedná o nejlepší analytické řešení.

Každý výskyt ze třídy Fyzická osoba bude obsahovat jako svoji kompozici výskyt ze třídy Adresa v roli Trvalé bydliště. Tato věta se napíše graficky v UML takto:



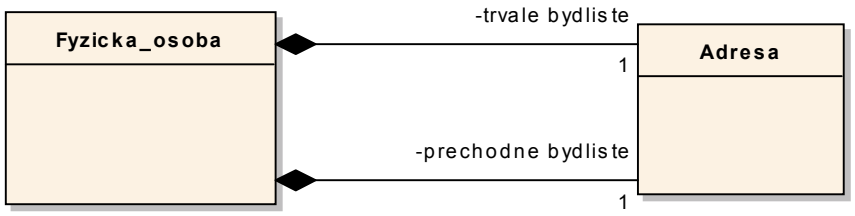
obrázek 19 Příklad na roli

V tomto případě lze diagram přečíst takto:

Každá instance ze třídy Fyzická osoba bude obsahovat jednu instanci ze třídy Adresa v roli Trvalé bydliště. Instance ze třídy Fyzická osoba používá třídu Adresa a to tak, že v ní „žije“ instance, a to v roli trvalé bydliště (tak ji „vidí“ instance ze třídy Fyzická osoba).

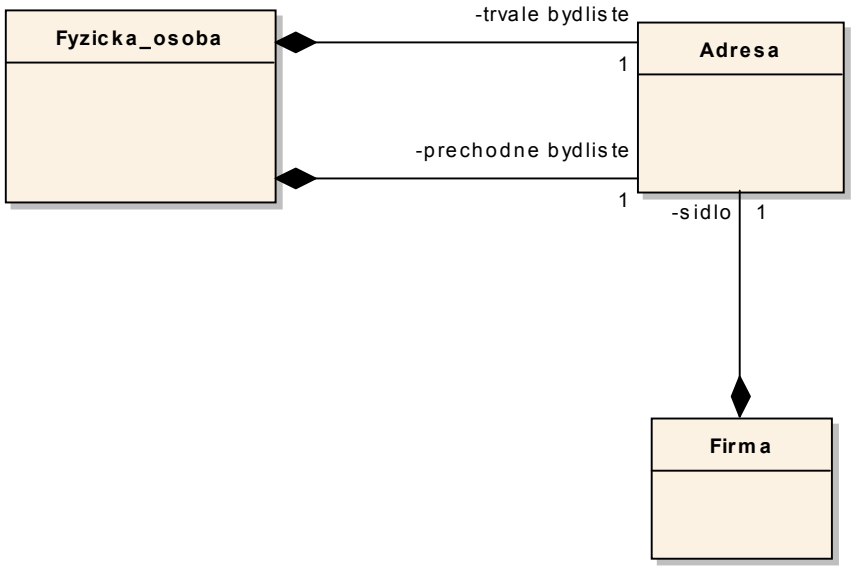
Díky kompozici instance trvalé bydliště ze třídy Adresa žije v řízení instance ze třídy Fyzická osoba a nijak jinak. Pokud se například daná instance ze třídy Fyzická osoba z evidence vymaže, potom nemůže „její“ instance Trvalé bydliště ze třídy Adresa zůstat „viset“ v systému a také odchází s daným výskytem ze třídy Fyzická osoba „do záhrobí“. Prvky vlastněné v kompozici se chovají jako části (odtud název kompozice), které neumějí žít jinak než jako části svých majitelů.

Ještě více vynikne význam role, pokud kromě instance trvalého bydliště zavedeme další evidované výskyty adres, například instanci přechodného bydliště z téže třídy Adresa. Model můžeme doplnit o další vztah:



obrázek 20 Dvě role třídy Adresa, přechodné a trvalé bydliště

Na předešlém obrázku obsahuje každý výskyt z třídy Fyzická osoba dvě instance ze třídy Adresa (pozor, nikoliv seznam se dvěma instancemi, ale dvě instance vedle sebe stojící). Jedna hraje roli trvalé bydliště, druhá hraje roli přechodné bydliště. Obě instance pocházejí z téže třídy Adresa (tj. obě instance převzaly ze třídy stejné vlastnosti), jejich konkrétní obsah je však jiný. Navíc může existovat také ještě další role třídy Adresa, například sídlo firmy:

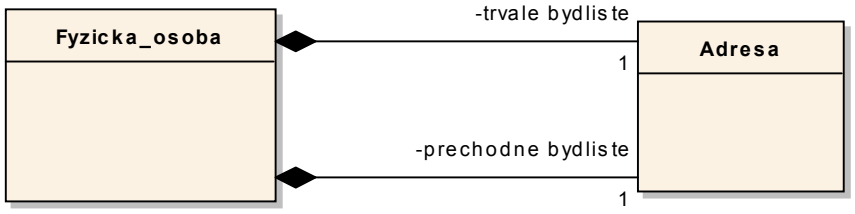


obrázek 21 Tři kompozice a tři role třídy Adresa

Předešlý diagram vyjadřuje tři myšlenky:

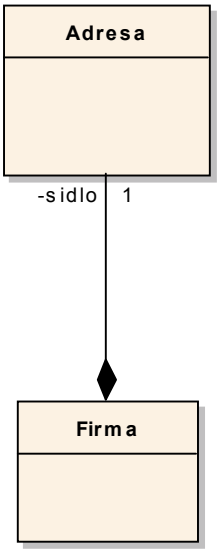
1) každá instance ze třídy Fyzická osoba obsahuje jednu instanci Adresy v roli trvalé bydliště,
2) každá instance ze třídy Fyzická osoba obsahuje jednu instanci Adresy v roli přechodné bydliště a
3) každá instance ze třídy Firma obsahuje jednu instanci Adresy v roli sídlo,

Podotkněme, že tato část modelu, kterou vyjadřuje předešlý obrázek se všemi třemi entitami (třídami) namalovanými do jednoho diagramu, nemusí být znázorněna povinně přesně takto, ale i jinak. Uvedené tři myšlenky lze namalovat pomocí vícero, třebas dvou nebo tří diagramů. Například první diagram by obsahoval pouze prezentační prvky pro třídy Fyzická osoba a Adresa a druhý by obsahoval prezentační prvky pro třídy Firma a Adresa (tj. tatáž třída se „opakuje“ na dvou diagramech). První diagram by pak mohl vypadat například takto:



obrázek 22 První diagram

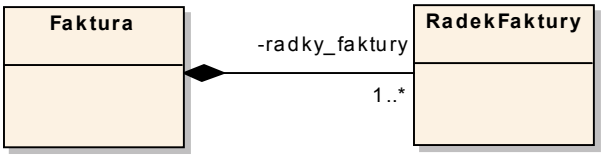
a druhý diagram (symbolicky řečeno „na jiné stránce“) by pak mohl vypadat takto:



obrázek 23 Druhý diagram

Je dobré si uvědomit, že předešlé dva obrázky chápané dohromady jsou plně ekvivalentní jednomu obrázku viz obrázek 21, protože třída Adresa znázorněná na obrázcích je v modelu jenom jedna. Záleží pouze na vůli autora, zda zvolí variantu „všechny prvky na jednom obrázku (diagramu)“ anebo rozdělí myšlenky do několika obrázků (diagramů).

U multiplicity „ku N“, tj. *, kdy instance drží v kompozici N prvků (seznam instancí), se také u třídy instancí vlastněných zavádí role. Tato role má v tomto případě význam názvu seznamu držených prvků. Volí se většinou množné číslo názvů těchto prvků (řádky faktury, služby klienta apod.). V diagramu se role znázorní stejně jako v předešlém příkladu s adresami, např. takto:



obrázek 24 Role radky_faktury označuje význam třídy - seznam řádků faktury

V tomto případě je role názvem celého seznamu řádků obsaženého ve faktuře.

4.11 Atribut a primitivní datový typ

Existuje zvláštní případ vztahu mezi informacemi, který sice svou povahou odpovídá kompozici s multiplicitou ku 1, ale v modelování v syntaxi UML se značí jiným způsobem. Jedná se o tzv. atribut. Je to takový vztah informace, ve kterém instance ze třídy A obsahuje instanci ze třídy B úplně stejně jako u kompozice ku 1, ale třída B je považována za tzv. ***obecně známý primitivní datový typ***. Chováním se jedná o úplně stejný typ vazby se stejným chováním jako kompozice ku jedné, ale vztah k instanci ze třídy B se zavádí nikoliv jako kompozice tříd, ale jako atribut ve třídě A. Jinak vše ostatní, co bylo řečeno o kompozici ku jedné, platí stejně také pro atribut. Z tohoto hlediska lze atribut chápat jako jednodušší zápis syntaxe kompozice ku 1 u jednoduchých typů informace.

V analytickém modelování se doporučuje zavést minimálně tyto datové typy atributů : *číslo, resp. krátké číslo a dlouhé číslo, řetězec a boolean.* Je možné tento seznam podle požadavků analytického modelování ještě rozšířit (například o typ *datum, money* apod.), případně dále upřesnit. Je však třeba dodržet určitá pravidla. Pokud se zavádí informace jako atribut, tak musí platit:

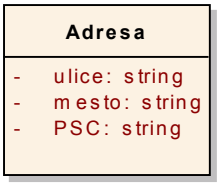
- daný typ informace nesmí být poplatný danému řešení problémové domény, tj. typ atributu musí být opravdu obecně známým, všude použitelným datovým typem
- násobnost vazby k atributu je ku 1. Pokud je násobnost *, tak v tom případě má kompozit právo na svou vlastní třídu. UML sice syntaxi „násobných atributů“ dovoluje, ale z hlediska analytického modelování a následného mapování do designu by násobné atributy vedly ke kolizím a proto se jí v analytickém modelování vyhýbáme. Poznámka: Tato možnost syntaxe násobných atributů v UML existuje pro možnost modelování obdoby datových struktur pole v designu (array apod.), v analytickém modelování se však zásadně nepoužívá.
- typ atributu nesmí reprezentovat informaci složenou z dalších informací (viz například třída Adresa obsahující další skladbu trojice Ulice, Město a PSČ). Analytické modelování přísně vyžaduje zavést pro takovouto informaci třídu a nikoliv primitivní datový typ a atribut.

Stručně řečeno, typ atributu musí být jednoduchým datovým typem - skalárem.

Při dodržení těchto pravidel je mapování do designu rychlé a efektivní s možností použití vzorů. Uvedené kolize při mapování do designu by vznikly díky tomu, že v případě použití složených atributů resp. násobných atributů by tyto atributy vyžadovaly ještě navíc další složité kroky mapování. Design totiž v konečném důsledku pracuje s „atomickými informacemi typu skalár“, jako jsou sloupce tabulek, atributy objektů, pole v stromové databázi apod.

Atribut se v UML zavádí přímo v daném CASE nástroji jako detail třídy. Název atributu odpovídá stejnému významu, jako byla role, tj. název instance v kompozici ku jedné a zmíněný datový typ má význam třídy v kompozicí ku jedné.

Pokud se vrátíme k našemu příkladu s kompozicí instancí typu Adresa, potom bychom při znalosti jak pracovat s atributy mohli tento příklad dále rozvinout. Uvnitř třídy Adresa zavedeme tři atributy typu řetězec (dále označujme tento typ jako string), a to atributy s názvy ulice, město a PSČ:

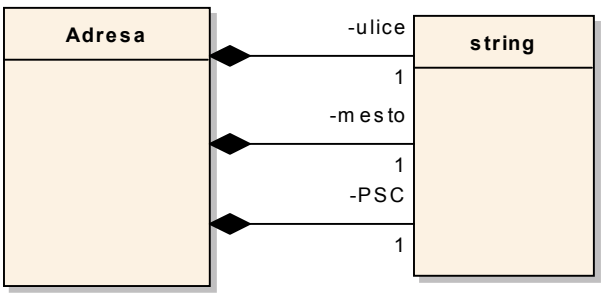


obrázek 25 Zavedení atributů u třídy Adresa

Předešlý obrázek lze číst tak, že každá instance ze třídy Adresa obsahuje tři atributy (tj. tři „své“ instance), všechny jsou typu string (ekvivalentně ze třídy string v kompozici ku jedné).

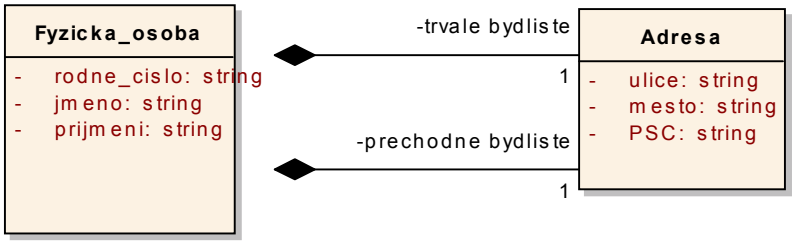
Poznámka: Také bychom měli udat délku těchto řetězců, zde v tomto příkladě jakože zatím ještě nezavádíme.

Následující zápis by teoreticky byl možný, přičemž chování prvků je v podstatě stejné jako v předešlém obrázku, ale nedoporučuje se pro vysoké znepřehlednění diagramů:

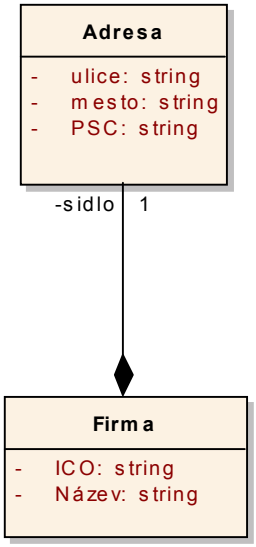


obrázek 26 Třída namísto typu u atributu se nedoporučuje,i když teoreticky možná je

Podobně bychom mohli začít doplňovat atributy také u dalších tříd, jako je Fyzická osoba a Firma apod.:



obrázek 27 Doplnění atributů u třídy Fyzická osoba



obrázek 28 Doplnění atributů u třídy Firma

Předešlé dva obrázky lze číst takto:

Každá evidovaná instance ze třídy Fyzická osoba obsahuje atributy rodné číslo, jméno a příjmení typu řetězec a obsahuje dvě instance, přechodné a trvalé bydliště typu (tj. ze třídy) Adresa, navíc každá instance ze třídy Firma obsahuje IČO a název typu řetězec a instanci sídlo typu Adresa, každá instance ze třídy Adresa obsahuje ulici, město a PSČ jako atributy typu řetězec.

Je dobré si uvědomit, že v tomto případě při kompozici je povaha vztahu „instance ze třídy Fyzická osoba obsahuje atribut rodné číslo typu řetězec“ a vztahu „„instance ze třídy Fyzická osoba obsahuje instanci trvalé bydliště ze třídy Adresa….“ úplně stejná a oba vztahy se chovají úplně stejně jako kompozice ku jedné. Například pokud se vymaže daná instance ze třídy Fyzická osoba ze systému, tak s ní odchází pryč jak její atributy se svými hodnotami, tak i obě dvě instance typu Adresa se svými hodnotami.

4.12 OBJECT (INSTANCE) MODEL - Objektový (instanční) model

V dalším výkladu budeme potřebovat zavést notaci modelu, který se nazývá OBJECT nebo také INSTANCE MODEL, nejčastěji překládaný jako objektový neboli instanční model. Jedná se o model, který se používá zejména v těch situacích při tvorbě analytických modelů, kdy je třeba ozřejmit nebo vysvětlit nějaký problém. Není proto divu, že se tento model používá zejména v raných fázích analýzy ve chvílích, kdy se model tříd teprve vyhledává. O tomto postupu bude pojednáno v příkladech použití objektového modelu.

Základními prvky CLASS MODELU jsou třídy a interakce mezi nimi, v objektovém modelu jsou základními prvky objekty neboli instance ze tříd a interakce mezi nimi. Je vcelku pochopitelné, že pokud porovnáme odpovídající model tříd a objektový model téhož IS, tak musí být v spolu souladu. Instance a jejich vztahy v instančním modelu odpovídají přesně svým třídám a vztahům mezi nimi v modelu tříd, protože tyto instance vznikají ze tříd a vztahy mezi objekty vznikají ze vztahů mezi třídami.

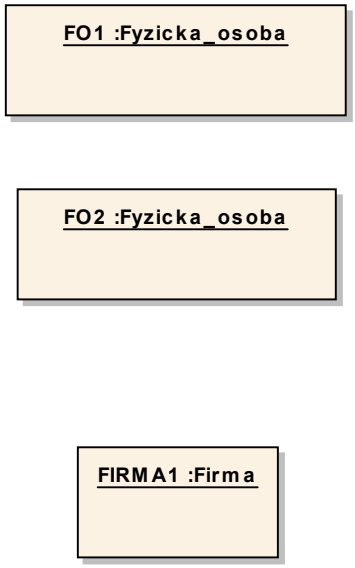
Je prospěšné si v této souvislosti připomenout, jak vlastně vidí analytik informační systém, viz znovu *obrázek 15 ANALYTICAL VIEW*. Zatímco model tříd znázorňuje nejvnitřnější statický genotyp aplikace tj. připravené třídy k rození instancí, objektový model ukazuje již samotné instance, které se z těchto tříd rodí. Z toho důvodu je objektový model chápán jako jeden z velmi mnoha možných příkladů již běžící evidence v „run-time“. Jeho diagramy ukazují, jaké by mohly v systému existovat analytické instance a vztahy mezi nimi pro daný uvažovaný příklad evidence. Objektový model je tedy třeba chápat pouze jako jednu z nekonečně mnoha možných situací evidence, která by mohla vzniknout díky statickému modelu tříd v pozadí aplikace. Proto většinou vysvětlení objektového modelu začíná slovy: „Nechť jsou evidovány dvě (tři, čtyři apod.) instance ze třídy XY, označme si je …“ atd.

Prvky objektového modelu, tj. objekty resp. instance, se značí pomocí obdélníků s názvy prvků, které se podtrhují. Pokud je známo z jaké třídy objekty (instance) pocházejí, tak UML umožňuje k tomuto prvku přiřadit také tuto třídu, oddělovačem je dvojtečka.

Je třeba si však vždy uvědomit, že označení prvků v objektovém modelu je námi zavedené označení prvků v modelu tj. instancí, a slouží pouze k odlišení těchto instancí v našem modelu (a k ničemu jinému).

Nyní si již můžeme ukázat, jak může vypadat instanční model v našem příkladu s adresami. Jako odpovídající model tříd zvolme příklad podle obrázku viz obrázek 21.

Nechť platí model podle zmíněného obrázku viz obrázek 21 a nechť v systému jsou evidovány nějaké dvě instance ze třídy Fyzická osoba (pozn.: může být, že jsou evidovány i další instance ze třídy Fyzická osoba, což nás nezajímá). Označme si tyto instance jako FO1 a FO2. Nechť je evidována instance ze třídy Firma (pozn.: opět může nastat, že jsou evidovány i další instance ze třídy firmy, což nás nezajímá). Označme si tuto instanci jako FIRMA1. Tuto evidenci bychom mohli vystihnout tímto diagramem:



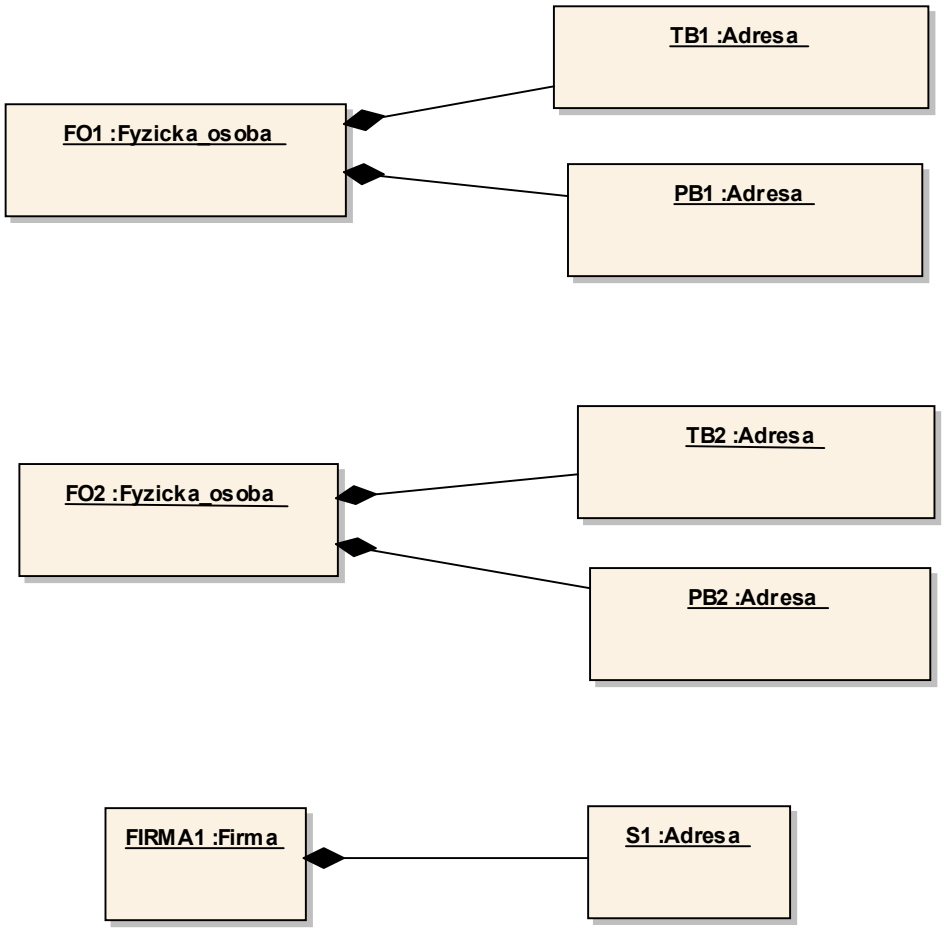
obrázek 29 Tři evidované instance, dvě ze třídy Fyzická osoba a jedna ze třídy Firma

Podle modelu tříd budou evidované instance FO1 a FO2 obsahovat každá dvě instance ze třídy Adresa, jedno trvalé a jedno přechodné bydliště, a instance FIRMA1 bude obsahovat jednu instanci ze třídy Adresa jako sídlo. Označme si tyto instance názvy prvků modelu po řadě jako TB1, PB1, TB2, PB2, S1 (pozn.: je třeba si uvědomit, že toto označení je pouze „naše označení“ pro účely tohoto diagramu!).

Musíme si ještě znázornit zmíněné vztahy kompozice, tj. obsažení instance v instanci a to v instančním modelu. V našem případě FO1 obsahuje v kompozici TB1 a PB1, FO2 obsahuje v kompozici TB2 a PB2 a FIRMA1 obsahuje v kompozici instanci S1. Je pravda, že máme sice zaveden vztah kompozice v modelu tříd, ale nemá jej zaveden v instančním modelu. V UML platí, že pokud je v modelu tříd zaveden vztah, který vede ke vztahu mezi instancemi (takovým vztahem je například námi uvažovaná kompozice), tak v instančním modelu mu odpovídá vztah, který se nazývá LINK. Je to vlastně instanční obraz vztahu mezi třídami. Podobně jako je objekt instancí třídy, tak LINK je obrazem - instancí vztahu mezi třídami. Platí analogie: Třída neboli CLASS nechává vzniknout instancím – objektům, podobně

vztah mezi třídami, který vede ke vztahu mezi instancemi, nechává vzniknout prvkům LINK jako spojnicím mezi objekty.

Zobrazme tedy náš příklad evidence s adresami takto (spojnice mezi objekty jsou chápány jako prvky LINK, značíme je graficky stejně jako kompozice):



obrázek 30 Instanční model i s adresami

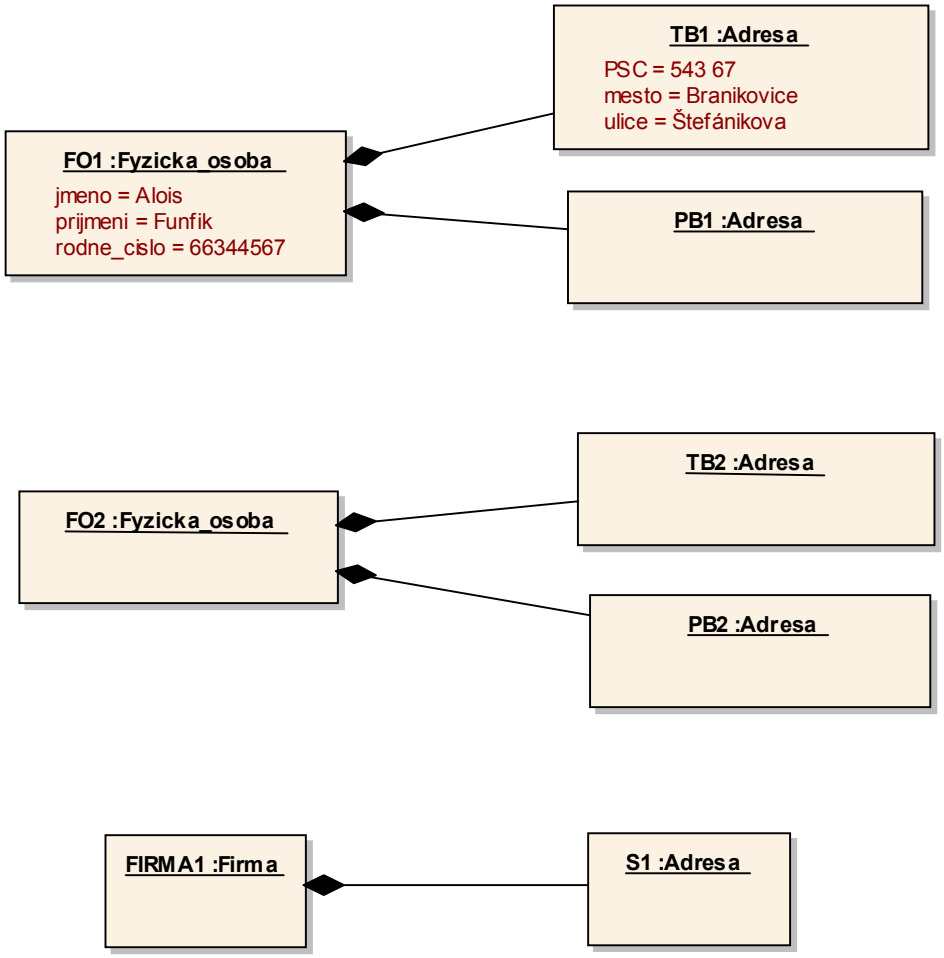
Uvedený obrázek není nic jiného, než grafický obraz již zmíněných vět: Instance FO1 obsahuje v kompozici instance TB1 a PB1, instance FO2 obsahuje v kompozici instance TB2 a PB2 a instance FIRMA1 obsahuje v kompozici instanci S1.

Tento příklad evidence kompozice byl zvolen proto, aby se na něm demonstrovala jedna ze základních vlastností kompozice, kterou bychom mohli nazvat „svůj k svému“. Je zřejmé, že odpovídající instance adres jsou v plném majitelství svých odpovídajících majitelů (tj. instancí FO1, FO2 a FIRMA1). Znamená to, že například

instance TB1 vznikne a zanikne v kontextu instance FO1 a nijak jinak než „v ní“ žít nemůže. Tato vlastnost je z kompozice zřejmá.

Avšak platí ještě jedna zajímavá skutečnost, která sice z již uvedených vlastností kompozice vyplývá, ale na první pohled není až tak zřejmá a mnohdy se nedomýšlí. Vysvětleme si tuto vlastnost kompozice „svůj k svému“:

Protože se jedná o instanční model, lze jednotlivým instancím přiřazovat již konkrétní hodnoty v atributech (samozřejmě opět jako příklad evidence). Vymysleme takovýto příklad a zobrazme jej v modelu. Nechť například v evidenci jsou zadány konkrétní hodnoty u FO1 a TB1 takto (pokud jsem se do někoho v příkladu trefil, nechť prosím promine):

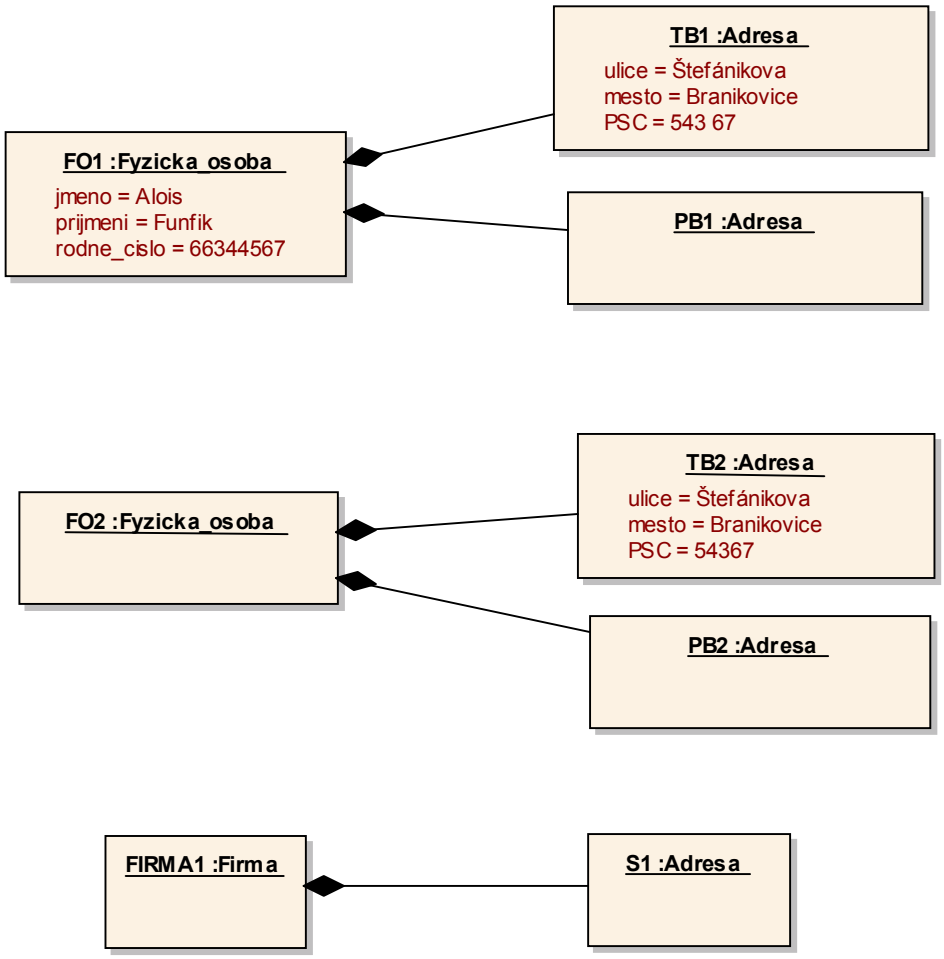


obrázek 31 V instančním modelu lze doplňovat konkrétní hodnoty atributů jako nějaký příklad evidence

Na předešlém obrázku jsme znázornili, že v evidenci se (jako příklad evidence) vyskytuje instance třídy Fyzická osoba označená v našem modelu jako FO1, která má konkrétní hodnoty rodné číslo, jméno a příjmení a tato instance obsahuje v kompozici instanci pro trvalé bydliště TB1 také s konkrétními hodnotami. Ostatní objekty (evidované instance) nemají v tomto příkladu atributy vyznačeny.

Otázka zní: Jak se projeví přímo v tomto příkladu a předešlém diagramu to, že další instance FO2 bude mít evidenci totéž trvalé bydliště jako instance FO1? Než odpovíme, tak buďme opatrní na povahu vazby kompozice.

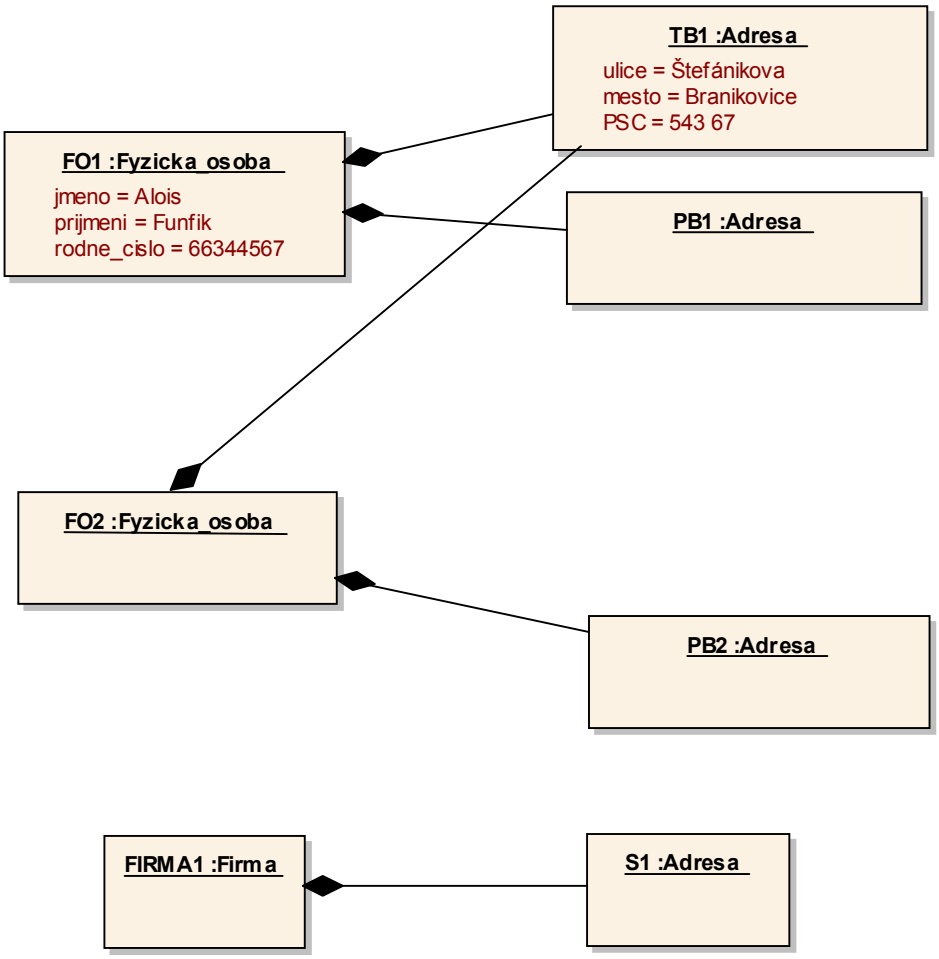
V tomto modelu (tak, jak je analytikem navržen) se to projeví tak, že nastane shoda hodnot atributů v adrese trvalého bydliště, tj. atributy instance TB2 budou mít stejné hodnoty jako atributy instance TB1. Tomu odpovídá následující obrázek:



obrázek 32 Shoda adres hodnotami

Pokud by nás v této chvíli napadlo, že předešlý model přece vede k redundanci adres při shodě hodnot adres, tak máme úplnou pravdu. To však cítíme jako určitou velmi nepříjemnou nevýhodu! Proto by další dotaz by mohl znít: „A nebylo by lepší, kdyby shoda adres byla vyjádřena nikoliv tak, že dvě instance mají stejné hodnoty, ale tak, že dvě osoby mají stejnou instanci adresy“?

Ten návrh by znamenal, že předešlý obrázek by se při shodě adres změnil nějak takto na tento:



obrázek 33 Návrh pro shodu adres, neshoda s modelem tříd

Návrh na předešlé obrázku je sice pochopitelný, skrývá však v sobě jeden „drobný háček“. Pokud bychom namalovali předešlý obrázek a přijali jej, tak se nejedná o shodu mezi modelem tříd, který nám odevzdal analytik a tímto instančním modelem! Předešlý obrázek se sdílenou instancí je totiž úplně jiné řešení, než jaké probíráme

v kapitole o kompozici, tj. nejedná se o návrh řešení adres pomocí kompozice ztvárněný podle obrázku viz obrázek 21.

Charakteristickým znakem kompozice je totiž to, že vlastněná instance (v tomto případě instance trvalého bydliště) patří daném majiteli a nikomu jinému a to vždy, od svého narození až po své vymazání, nemůže tedy zásadně dojít ke sdílení instancí mezi dvěma majiteli. Sdílení instancí není kompozice, ale jiný vztah, který jsme ještě nebrali.

Pokud tedy analytik navrhl řešení adres tak, jak je zobrazeno na obrázku viz obrázek 21, tak tím dal najevo, že s každou instancí ze třídy Fyzická osoba vznikne další pouze její a nikoho jiného instance trvalého bydliště ze třídy Adresa bez ohledu na hodnoty v této adrese. Teď nerozebíráme, zda se jedná o analyticky lepší nebo horší řešení, prostě takto to analytik navrhl. Právě tímto chováním (které vede i k možné redundanci) máme na mysli onu vlastnost „svůj k svému“. S trochou nadsázky lze kompozici přirovnat k příloze v mailu, kdy každý mail má svou přílohu, i když může jít u dvou mailů o stejný obsah příloh.

Je třeba si uvědomit, že návrh na předešlém diagram se „stejnou instancí pro dvě osoby“ neodpovídá kompozici a vyjadřuje jiný vztah mezi třídami než je kompozice (další vztahy budeme brát za chvíli). Pokud se nám tedy analytikův návrh nelíbí, tak bychom za ním měli jít a vůči němu vznést námitku. Rozhovor by potom mohl znít nějak takto:

„Franto (Pepo nebo podobně), víš že jsi navrhl adresy jako kompozice ku jedné?“

„Jasně.“

„A víš, že to vede k opakování adres, když jsou shodné?!“

„Jasně.“

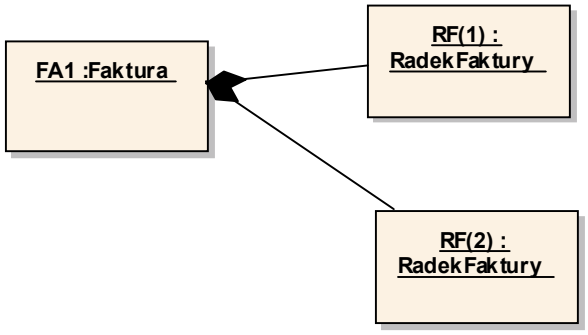
„A nebude to v tom našem informačním systému vadit?“

„Jasně, že ne…(a následuje opravdu relevantní a logické zdůvodnění)“…

Anebo analytik Franta (Pepa nebo podobně) zvolá „Proboha, to mi nedošlo!“ a celé to předělá na jiný vztah, který jsme ještě nebrali.

Je třeba podotknout, že z praxe znám systémy, kdy je opravdu výhodné řešení adres s kompozicí a znám systémy, kdy je absolutně nevhodné. K této diskusi se vrátíme, až probereme další vztahy, které potřebujeme pro analytické modelování.

Jako další si uvedeme příklad na znázornění objektového modelu při kompozici ku N:



obrázek 34 Příklad instančního modelu kompozice ku N

Mezi tímto obrázkem s řádky a předešlými obrázky s adresami je určitý rozdíl v tom smyslu, že u příkladu s adresami stály dvě instance (trvalé a přechodné bydliště) ve fyzické osobě „vedle sebe“ jako dvě od sebe názvem odlišitelné instance. V druhém příkladu s řádky faktur na předešlém obrázku patří dvě instance do jednoho seznamu a (jako jiný příklad evidence) by jich mohlo být například 3, 4 (obecně N instancí).

Poznámka: Samozřejmě, že je možné řešit adresy také pomocí seznamu, ale k tomu opět potřebujeme další vztahy mezi třídami, které nemáme probrány.

4.13 Mapování vztahu kompozice ku jedné do relační databáze

Předešlé úvahy ohledně vztahu kompozice mají velice konkrétní dopad na to, jak bude následně vypadat program. Předešlé úvahy nejsou nějakými „obecnými a mlhavými“ představami, ale jedná se o konkrétní zadání pro designéra.

Ukažme si jako příklad konkrétní ukázky přechodu od analytického modelu do designu pro případ použití relační databáze.

Mapování do relační databáze provádí designér na základě analytického modelu tříd. Vyjdeme z definované třídy Fyzická osoba. Na základě její existence a existence jejích tří atributů designér navrhuje tabulku například s názvem TFYZ_OSOBA, tabulka má tři sloupce: rodnecislo, jmeno a prijmeni. Obrazem třídy je tabulka, která se chová jako „typ“ v relační databázi. Její záznamy jsou její instance, například:

TFYZ_OSOBA

rodnecislo	jmeno	prijmeni
66344567	Alois	Funfik

obrázek 35 Tabulka TFYZ_OSOBA a jeden konkrétní záznam

Předešlý obrázek znázorňuje jeden záznam v tabulce TFYZ_OSOBA, tabulka má sloupce s názvy rodnecislo, jmeno a prijmeni. Jeden konkrétní záznam (na obrázku šedý) těmto sloupcům dává konkrétní hodnoty.

Porovnejme si nyní předešlý *obrázek 35* a *obrázek 30.* Úmyslně jsme zvolili stejné hodnoty jak ve sloupcích předešlého obrázku , tak v atributech analytické instance FO1 (viz *obrázek 31* V instančním modelu lze doplňovat konkrétní hodnoty atributů jako nějaký příklad evidence). Dá se dokonce říci, že předešlý *obrázek 35* a *obrázek 30* mají k sobě velmi blízko: Z hlediska evidované osoby FO1 popisují tutéž situaci, ale každý ji zobrazuje na jiné úrovni abstrakce. Uvedený *obrázek 35* ukazuje evidovanou instanci s hodnotami rodného čísla, jména a příjmení „66344567, Alois, Funfik“ tak, jak si ji představujeme v databázi, tj. v konkrétním databázovém prostředí. Podobně *obrázek 30* popisuje tutéž instanci FO1 v obecnější rovině, tj. ve vyšší abstrakci nezávisle na tom, zda se jedná o databázi, soubory, proměnné apod.

Je třeba ještě podotknout, že navíc jak ukazuje *obrázek 16*, jedná se pouze o jednu „polovinu“ z celého mapování z analytického modelování do designu hybridního systému, protože instance FO1 je na předešlém obrázku s tabulkou mapována

pouze do své persistentní části (do dat v RDB). Nad tabulkami musí pracovat ještě nějaká aplikace, tj. musí být navrženy ještě funkce a proměnné, resp. objekty, následně nějaké SQL příkazy atd. Tyto prvky z aplikace musí být také mapovány z analytického modelu.

Nyní je otázkou, jak bude realizována relace mezi tabulkami, která vznikne jako obraz kompozice ku jedné. Pro vysvětlení použijeme zmíněný příklad s adresami. Nechť tedy analytik navrhl řešení adres pomocí kompozice ku jedné a je platný diagram zobrazený například pomocí obrázku viz *obrázek 21*. Jak toto zadání zrealizuje designér konkrétně pro relační databázi, jinak řečeno jaké vzniknou tabulky a jaké relace mezi nimi?

V tomto případě má designér na výběr ze dvou známých postupů mapování do RDB.

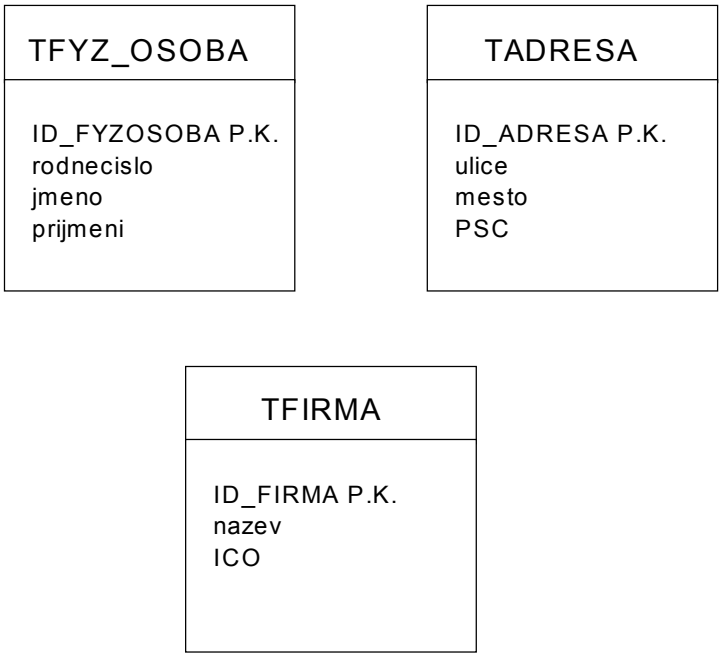
4.14 Čisté mapování kompozice ku jedné do RDB

Jako první je nejjednodušší způsob mapování, které budeme nazývat „ve třídách jedna ku jedné“ nebo také jako „čisté mapování“. Jeho princip je jednoduchý: Mapuje se tak, že co analytická třída, to tabulka. V tom případě počet tabulek odpovídá počtu tříd.

V našem případě, pokud se podíváme na *obrázek 21*, tak vzniknou díky této části analytického modelu tři tabulky. Nazvěme je TFYZ_OSOBA, TADRESA a TFIRMA. Odpovídající sloupce nazvěme například takto:

- tabulka TFYZ_OSOBA má sloupce rodnecislo, jmeno, prijmeni,
- tabulka TADRESA má sloupce ulice, mesto, PSC
- tabulka TFIRMA má sloupce nazev, ICO.

Navíc každou tabulku vybavíme primárním systémovým klíčem, který identifikuje daný záznam. Názvy těchto primárních klíčů zvolme jako „ID plus podtržítko plus název dané entity“ (např. ID_ADRESA). Tabulky mohou vypadat v této fázi například takto (pozn.: zkratka P.K. označuje zmíněný primární klíč neboli primary key):



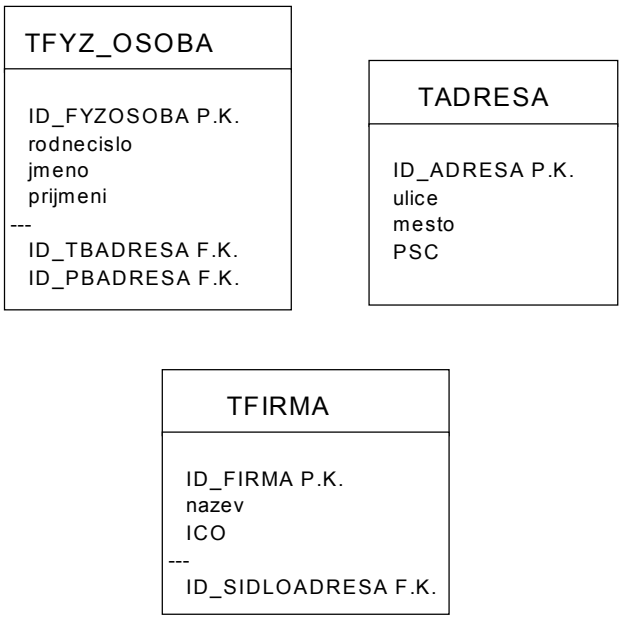
obrázek 36 Návrh tabulek, zatím bez cizích klíčů

Musíme také v databázi vyjádřit analytikovy myšlenky z analytického modelu tříd, které znějí:

1. každá instance ze třídy Fyzická osoba obsahuje jednu instanci Adresy v roli trvalé bydliště,
2. každá instance ze třídy Fyzická osoba obsahuje jednu instanci Adresy v roli přechodné bydliště a
3. každá instance ze třídy Firma obsahuje jednu instanci Adresy v roli sídlo

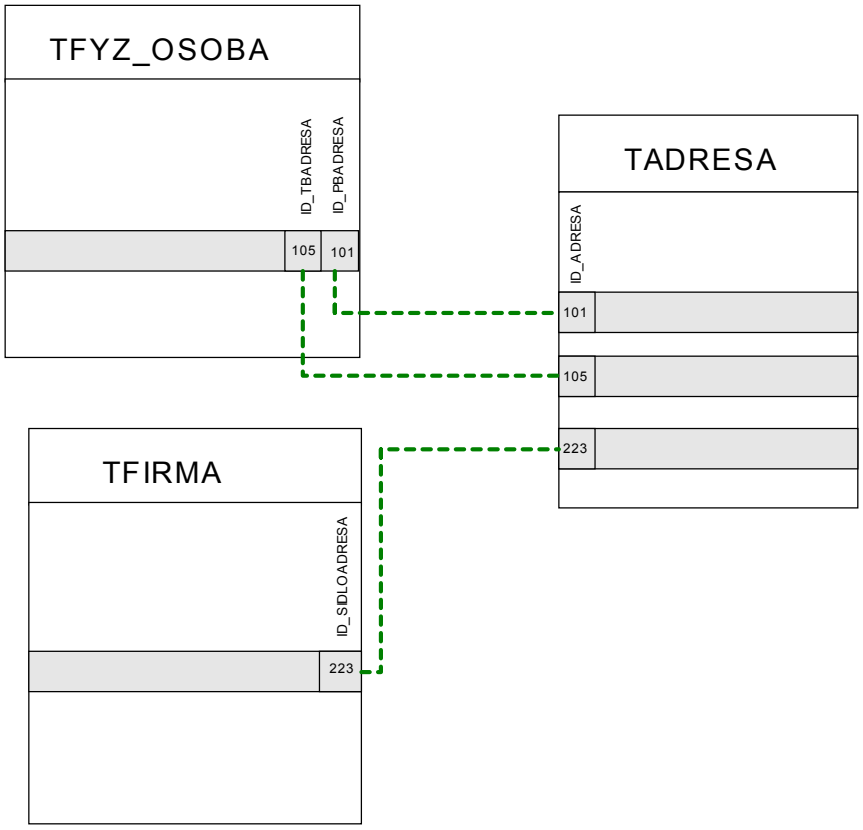
Vzniká otázka, kde se objeví cizí klíče, které provazují dané záznamy podle těchto pravidel.

V tomto případě při mapování „ve třídách jedna ku jedné“ (tj. čisté mapování) putuje klíč z tabulky TADRESA do tabulky TFYZ_OSOBA a to hned dvakrát pro obě instance trvalého bydliště a přechodného bydliště, současně z tabulky TADRESA do TFIRMA putuje klíč pro jednu instanci sídla. Jinak řečeno, tabulka TADRESA nabízí svůj primární klíč jako cizí klíč do jiných tabulek, což vyjadřuje „kam je záznam vložen“. Název cizího klíče se zvolí jako složený název z názvu role a z názvu entity, tedy například takto:



obrázek 37 Tabulky i s cizími klíči

Záznamy se přes tyto cizí klíče provazují tak, že daný záznam v tabulce fyzické osoby TFYZ_OSOBA má trvalé bydliště jako záznam v tabulce adres tam, kde je v tabulce adres TADRESA záznam adresy s klíčem rovným ID_TBADRESA, podobně pro přechodné bydliště a sídlo. Příklad takové evidence:



obrázek 38 Záznamy v relační databázi kompozice ku jedné s "čistým mapováním"

Opět se vrátíme k obrázku viz *obrázek 30,* který zobrazuje instanční model. Dobře si oba obrázky, jak předešlý, tak *obrázek 30*, prohlédněme. Jejich podoba je úžasná, ale není náhodná. Malý rozdíl je v tom, že na obrázku *obrázek 30* jsou dvě instance fyzické osoby, kdežto na předešlém obrázku pouze jedna, ale to není podstatné. Protože z analytického modelu tříd vznikly tabulky a z instancí záznamy, není divu, že oba obrázky jsou si velmi podobné. Dokonce s trochou nadsázky se dá říci, že předešlý obrázek lze chápat jako „instanční model vyjádřený speciálně v RDB“.

Samozřejmě analytik se těmto obrázkům speciálním pro nějaké prostředí vyhne a vidí aplikaci na své abstraktní úrovni analytického modelování nezávisle na implementačních podrobnostech, tj. vidí aplikaci jako *obrázek 30*. Je však dobré, aby i analytik viděl a rozuměl tomu, jak designér jeho myšlenky v designu ztvární.

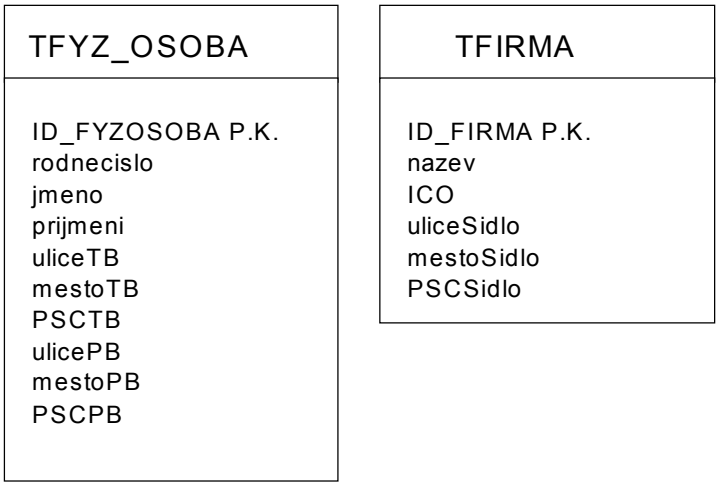
4.15 Mapování kompozice ku jedné do RDB s rozpuštěním kompozitu

Existuje také druhý možný způsob mapování kompozice ku jedné do relační databáze, který si nyní ukážeme. Tento způsob mapování je příznačný pro relační databázi a souvisí s odstraněním vazby mezi tabulkami. Tato vazba (v našem případě vazba s tabulkou TADRESA přes klíč ID_ADRESA) může vést v některých scénářích ke zpomalení zpracování díky povaze relační databáze. Většinou se takovým kritickým scénářem stává proces nějakého hromadného zpracování. Uvedená vazba JOIN mezi tabulkami může způsobit zpomalení a následný problém. Je třeba však upozornit na tu skutečnost, že mnohdy uvedená pomalost bývá zapříčiněna nikoliv samotnou vazbou, ale neznalostmi designéra, který chybně navrhne proces zpracování, například v pořadí dotazů ve zpracování, ve špatném indexování apod.

Poznámka: Jako analytik jsem byl svědkem situace, kdy jeden „designér amatér“ navrhl určité zpracování údajů v databázi (noční uzávěrka bankovního účetního systému), přičemž druhý „guru designér“ navrhl tentýž proces „trochu jinak“, s jinými indexy, s jiným pořadím v SQL příkazech a podobnými finesami, ale bez zásahu do struktury tabulek. Analyticky totéž zpracování se těmito zásahy zrychlilo o tři řády.

Pokud designér zvolí mapování „rozpuštění kompozitu ku jedné“, měl by mít pouze jediný důvod: Odstranění technologických problémů, které vznikají vazbou (JOIN) mezi tabulkami, kdy všechny známé jiné pokusy selhaly.

Vyjděme opět z analytického modelu na obrázku *obrázek 21*. Designér neprovede mapování ve třídách jedna ku jedné a entitu Adresa v tabulkách tzv. rozpustí. Vznikne tak zajímavá situace: V modelu tříd od analytika se tato třída Adresa vyskytne, ale v tabulkách nenajdeme žádnou tabulku TADRESA. Všechny sloupce této tabulky se přesunou do odpovídajících tabulek namísto původního jednoho cizího klíče. Tabulka TADRESA se doslova rozpustí do ostatních tabulek. Odtud jsme zvolili název tohoto mapování do RDB jako „Rozpuštění kompozitu ku jedné v RDB“. Názvy těchto sloupců v odpovídajících tabulkách budou dány jednak rolemi v analytickém modelu a jednak budou dány názvy entit, odkud původně pocházejí. V našem příkladu namísto tří tabulek dostaneme pouze dvě tabulky, ale v nich se objeví větší počet sloupců, např. takto:



obrázek 39 Mapování do RDB s rozpuštěnou entitou Adresa

Díky tomu, že tabulka TADRESA jako taková neexistuje, tak uchycením (slangově „fečnutím“) jednoho záznamu v tabulce TFYZ_OSOBA můžeme rovnou bez vazby přes klíč načíst všechny hodnoty včetně hodnot adres. Při mapování do RDB „ve třídách jedna ku jedné“ bychom museli pro načtení údajů adres jít přes vazbu klíčů s klauzulí WHERE do vedlejší tabulky. Podobně totéž platí i pro záznam v tabulce TFIRMA a pro jeho záznam sídla v adresách.

Zdá se, že jsme „něco získali“ ale pravdou je, že jsme také „něco ztratili“. Je třeba vědět, že základní věc, kterou jsme tímto postupem ztratili, je opětovná použitelnost a to se všemi nepříznivými důsledky z toho plynoucími.

Porovnejme oba možné postupy mapování, jejich výhody a nevýhody. Začneme jednoduchou úvahou. Pokud designér navrhne „čisté mapování“ a tabulka adres TADRESA existuje, tak jednoduchá otázka zní: Kolikrát se bude programovat INSERT do tabulky TADRESA? Odpověď zní: Pro všechny případy pouze jednou. Dokonce pokud je systém navržen v designu poměrně dost čistě (například v aplikaci pomocí OOP s použitím návrhových vzorů), tak vše, co se týče adres, bude programováno pouze jednou a jednotlivé prvky softwaru, které adresy používají (jako jsou Fyzické osoby, Firmy atd.), budou tyto funkcionality pouze volat. Dovedeme si představit například takovouto konstrukci: Je naprogramována část formuláře (například jako FRAME apod.), který obsahuje tři editační pole pro ulici, město a PSČ. Za tímto formulářem je přiřazena nějaká aplikační logika (například objekt nebo skupina funkcí) a úplně v pozadí se nacházejí prvky datové a tabulka TADRESA. Použití této naprogramované části systému je možné například takto: Do formuláře pro editaci fyzické osoby je tento FRAME vložen jako instance dvakrát, do formuláře pro editaci firmy jednou, jinak řečeno, při editaci fyzické osoby se použijí dvě instance z tohoto FRAME, u firmy jedna. Po vyplnění se zavolá funkcionalita uschovaná za těmito instancemi.

Pokud se však tabulka rozpustí, tak se každá entita musí o svou adresu resp. adresy postarat sama. Všude se budou opakovat skupiny polí, například v SQL příkazech ve smyslu „INSERT VALUES zase další pole ulice, město, PSC INTO TABLE MOJE_TABULKA“.

Jak víme, existuje však ještě horší možný důsledek opouštění opětovné použitelnosti, o něm pojednává další kapitola.

4.16 Ztráta transparence systému při optimalizaci návrhu

Dalo by se namítnout, že opakování prací uvedené v předešlé kapitole, které vzniká při rozpuštění tabulky, přece není až takovou hrůzou. Daná situace se přece dá při troše úsilí zvládnout! (pozn.: Oblíbená to věta manažerů ☺ …). Odpověď na tuto námitku zní: Ano, lze ji zvládnout, ale za určitého předpokladu.

Z předešlých kapitol víme, že nejhorším důsledkem opuštění opětovné použitelnosti není samotné opakování prací, ale daleko horším následkem je možná ztráta transparence systému. Ukažme si zde konkrétně, jak taková ztráta transparence může vypadat i se svými následky.

Představme si tu situaci, kdy analytik přijde a prohlásí: „Pánové, je mi to hrozně líto, ale náš konzultant a zákazník se nějak trochu pozapomněl. V té adrese má být ještě okres. Konstrukce okresu je taková a taková (analytik předloží upravený model). Tak to tam dodělejte…“. Z hlediska analytika vypadá problém jednoduše, ale může se stát, že vývojáře mohou díky tomuto požadavku čekat bezesné noci právě díky ztrátě transparence systému.

Pro další práce mohou nastat tyto tři situace, z nich třetí je pro zásah do systému katastrofou:

1. Situace velmi příznivá: Tabulka TADRESA nebyla rozpuštěna a existuje analytický model tříd. Protože systém je navržen poměrně dost čistě (například pomocí OOP a pomocí design patterns) a vývoj je dobře zdokumentován (včetně fáze analytického modelování), tak se přesně ví, ze které analytické třídy tabulka pochází. Zásah se týká pouze samotné adresy a nikoliv toho, kdo tuto adresu používá. Změna je cílená a není bolestivá. V našem příkladu se pouze vymění jak uvedený FRAME, tak za ním funkcionalita včetně upravené tabulky. Každý volající tuto změnu buď nepocítí anebo (při určitých drobných nečistotách) pouze minimálně.

2. Situace stále ještě příznivá, ale pracná: Tabulka TADRESA byla rozpuštěna, ale existuje analytický model tříd. Z toho důvodu se ví velmi přesně, kam všude byla tato tabulka rozpuštěna a kdo a kde používá opakující se sloupce. Vydá se metodický pokyn, jak opravit opakující se adresy. Změny se sice pracně, ale cíleně provedou, většinou bez chyb a nikoliv chaoticky. Práce jsou však náročnější a pro dokumentaci složitější.
3. Situace vedoucí ke katastrofě při vývoji systému: Tabulka TADRESA je rozpuštěna a analytický model tříd neexistuje. Nikdo neví, kde všude se tyto sloupce opakují. V týmové práci a při předávání softwaru mezi pracovníky neexistuje relevantní dokumentace, která by vedla k systematickému vyhledání zmíněných sloupců adres v databázi. Jediný způsob, jak najít opakující se sloupce adres, je pracně je vyhledávat a intuitivně odhadovat, zda jsou to ony nebo ne. Zásah se neprovede systematicky, ale náhodně. Provede se pouze u těch skupin sloupců, které se najdou resp. na které se narazí při testování anebo na co upozorní sám velmi nespokojený uživatel („Však jsem vám to říkal, to už je potřetí, co jsem narazil na adresu bez okresu!“). V některých případech se omylem zasahuje do sloupců, kterých se tato změna netýká, protože názvy sloupců adres jsou pochopitelně různé, navíc vymyšlené a dodané od kolegů, z toho důvodu jsou v návrhu databáze obtížně čitelné. Systém není transparentní, má v sobě chyby, které se velmi těžko hledají. Všimněme si, že ať chceme nebo ne, tato varianta je vlastně cestou do zmíněného tunelu.

Rozeberme si blíže situaci třetí, protože ta je pro nás velmi zajímavá: Existuje sice návrh v designu, ale není čitelný a transparentní anebo pouze velmi obtížně. Otázkou je, proč tomu tak je? Odpověď na tuto otázku je mimo jiné jedním z mála světýlek, které nás vyvedou z metody řízení projektů nazvané jako TUNEL.

Problém spočívá v tom, že samotný návrh v designu v sobě vždy obsahuje „dvě logiky“. První logika spočívá v analytickém návrhu, který udává„o co tam v podstatě jde“ (v našem příkladu: „jde o přidání okresu v adrese“). Tato logika je myšlenkově čistá a transparentní, protože na úrovni abstrakce analytického modelování je pojem „adresa“ jako typ (třída) pouze jedním neopakujícím se pojmem. Vskutku, pokud přišel zákazník neboli konzultant a řekl: „Pardon, ale v adresách má být ještě okres…“ tak, aniž by uvažoval pomocí UML, měl na mysli instance ze jedné třídy adresa.

Druhá logika spočívá v technologickém postupu, který byl použit, v našem případě „Rozpuštění kompozitu v RDB“, konkrétně rozpuštění tabulky TADRESA mezi ostatní tabulky. Pokud bychom použili a zdokumentovali obě tyto logiky, tak systém zůstává transparentní. Zdokumentuje se „co se analyticky vymyslelo“ a „jak se to v designu realizovalo“.

Pokud ovšem odevzdáme rovnou výsledek designu (v tomto případě jako návrh databáze), nastává problém. Obě logiky se promíchají dohromady a výsledky práce vývojářů najednou postrádají transparenci a logiku, nejsou čitelné.

Mapování, které neodpovídá „čistému“ mapování ve třídách jedna ku jedné, se někdy nazývá optimalizace. Jak vidět, jedná se o název troch vzletný, osobně bych jako analytik nazýval optimalizaci raději jako „degenerace“, protože původní „čistý“ analytický model se v mapování projeví nějakými sekundárními zásahy. Optimalizace se pochopitelně provádí za nějakým účelem a přináší tedy nějaký technologický zisk (např. rychlost, obsazení paměti apod.). Na druhé straně každá optimalizace vždy a všude vede k nějakým ztrátám. Ukazuje se, že se vždy jedná o ztrátu opětovné použitelnosti se všemi důsledky.

Uvedený příklad ukazuje ještě na jednu zajímavou vlastnost: Pokud použijeme „čisté“ mapování bez optimalizace, tak se dokumentace velmi zjednodušuje. V tom případě totiž existuje přímá a rovná cesta mezi analytickým modelem a designem. Pokud se však použije některá z technik optimalizace (známe zatím „Rozpuštění kompozitu ku jedné v RDB“), tak se na dokumentaci kladou mnohem vyšší nároky a musí se zdokumentovat i koho všeho se tato optimalizace týká. Z toho důvodu se někdy hovoří o vysoké transparenci OOP systémů (pokud jsou dobře navrženy). Důvod je prostý: Čím více je aplikace v designu „čistěji“ objektově navržená, tím „přímější“ je cesta mapování od analytického modelu k designu. Navíc i další zásahy do systému jsou cílené a flexibilní (pokud se použijí správné konstrukce z design patterns).

Je ještě třeba podotknout, že v mnoha firmách, které putují TUNELEM, se pojem optimalizace ani nezavádí, protože návrhy databáze se provádějí přímo bez jakýchkoliv analytických modelů. Struktura databáze se přímo a rovnou „degeneruje“, nevědomky a bez rozmyslu rovnou při počátečním návrhu. Mohu z vlastní zkušenosti konzultanta potvrdit, že v mnoha případech jsou v těchto firmách návrhy databází s těmito „ad hoc“ postupy spíše paskvilem bez jakékoliv rozumné myšlenky. Používá se metoda „kam se co dá, tam to je“ bez ohledu na logiku analytických tříd a jejich vztahů. Vznikají velmi podivné tabulky s podivnými sloupci, tvoří se slepence a nelogické vazby. Takovýto systém se nejen velmi těžko vyvíjí, ale ještě hůře udržuje při životě. O následných požadovaných úpravách raději ani nemluvit.

Řešení problému ztráty transparence spočívá v oddělení obou zmíněných logik: Je třeba mít k dispozici jednak analytický model (tj. v tomto případě analytický model tříd), také postup mapování, nejlépe pomocí vzoru (známe zatím jeden vzor, „Rozpuštění kompozitu ku jedné v RDB“) a výsledek jako samotný design. Potom ztráta transparence nehrozí.

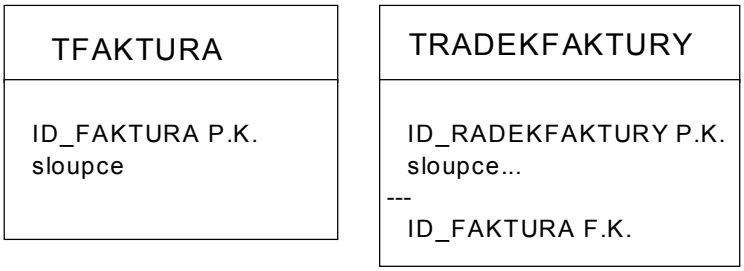
Poznámka: Uvedený vzor optimalizace „Rozpuštění kompozitu ku jedné v RDB“ je svým způsobem v původní teorii databází znám a odpovídá jednomu z postupů, který je opačný k normalizaci databáze 3. stupně. Někdy je tento postup v teorii databází nazýván jako „denormalizace“. Podoba našeho postupu „Rozpuštění kompozitu ku jedné v RDB“ s postupem denormalizace spočívá v té myšlence, že

vskutku správný a bezchybný analytický model tříd mapovaný jedna ku jedné do RDB odpovídá v tabulkách nakonec silně normalizované databázi 3.stupně. Důvodem této shody výsledku mapování a silné normalizace je princip opětovné použitelnosti zavedený v analytickém modelování jako podmínka nutná. Každý pojem má v AM právo na život, z čehož při mapování 1:1 do RDB vznikne silná normalizace relační databáze 3.stupně.

4.17 Mapování kompozice ku N do RDB „čisté“

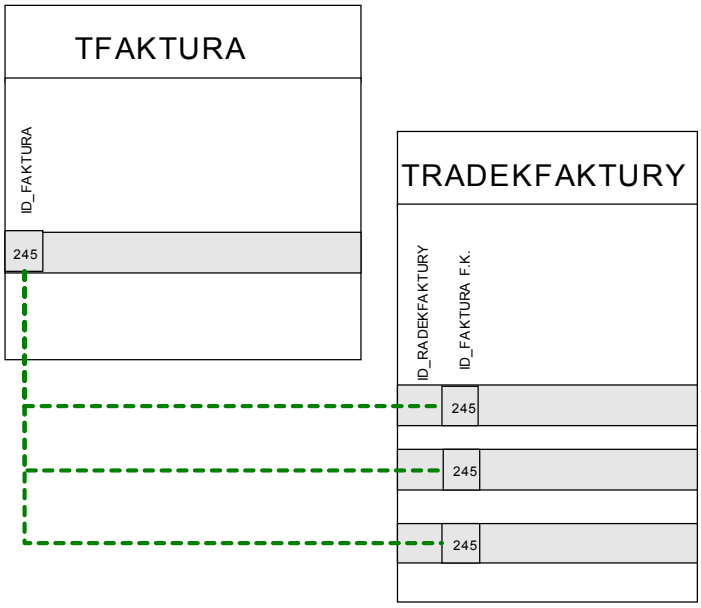
V předešlé kapitole byl popsán způsob mapování vztahu kompozice ku jedné do relační databáze. Nyní si ukážeme možné postupy mapování do RDB také pro kompozici ku N. Jako vzorový příklad si vybereme model s fakturou a jejími řádky.

Nechť tedy analytik určil vztah mezi analytickými třídami podle obrázku *viz obrázek 24*. Nejčastější způsob mapování uvedeného vztahu do relační databáze se provádí jako „mapování čisté 1:1“. Vytvoří se dvě tabulky, v tomto případě TFAKTURA a TRADEKFAKTURY a provážou se přes klíč tak, že v tabulce TRADEKFAKTURY se objeví jako cizí klíč primární klíč z tabulky TFAKTURA, např. takto:



obrázek 40 Mapování kompozice ku N do RDB 1:1

Pro představu je možné si zobrazit také „instanční model v RDB“, tj. jak by mohly vypadat provázané záznamy v tabulkách:



obrázek 41 Záznam nějaké konkrétní faktury a její odpovídající 3 řádky

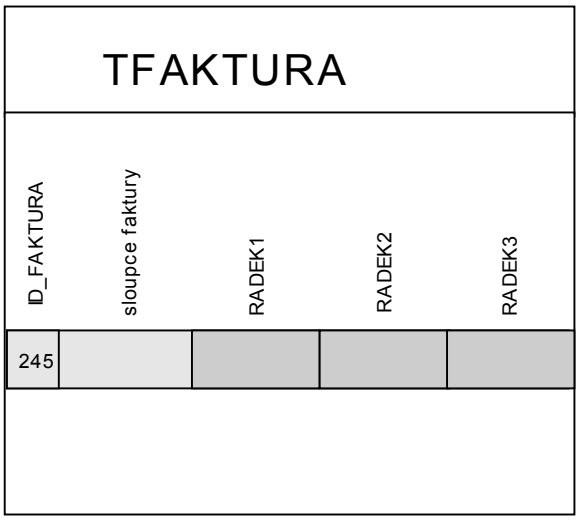
Není třeba znovu zdůvodňovat podobu mezi tímto obrázkem a obrázkem viz *obrázek 34*. Pokud pomineme drobnost, že na jednom jsou dvě instance a na druhém tři, tak rozdíl mezi nimi spočívá v tom, že předešlý *obrázek 41* je konkretizací obrázku viz *obrázek 34* speciálně pro pohled na záznamy v databázi.

4.18 Mapování kompozice ku N do RDB vzorem „Rozpuštění seznamu kompozitů v RDB“

Je možné provést i další možné mapování kompozice ku N do RDB s optimalizací, ale upřímně řečeno, nepoužívá se často, dokonce jsem s tímto přístupem setkal v praxi pouze výjimečně (doslova pouze jednou).

Princip této poměrně dost drastické optimalizace spočívá v tom, že tabulka kompozitu seznamu se nezavede (v našem příkladu neexistuje TRADEKFAKTURY) a její původní záznamy, které jsou pod sebou, se umístí za sebe do řady s opakujícími se sloupci. Je zřejmé, že tímto musí být počet prvků v seznamu kompozitu omezen, protože dynamická vazba ku N se převedla na statickou vazbu s opakovanými sloupci. Řádky faktury se takto většinou neřeší, protože možný maximální počet řádků je pro opakování sloupců ještě stále velký (řádově desítky).

Představme si však, že by platilo omezení, že řádků faktury může být v evidenci maximálně 3. Drastická optimalizace by pak mohla vypadat například takto:



obrázek 42 Mapování "Rozpuštění seznamu kompozitů v RDB"

Řádky se takto vyskytují přímo v majitelské tabulce a sloupce pro řádky se v tomto případě 3krát opakují (ve faktuře může být jeden, dva nebo maximálně tři řádky). Tento návrh sice urychluje práci se záznamy, ale optimalizace takto provedená je hodně drastická a opravdu výjimečná. Asi nemusíme zdůrazňovat, co se odehraje při následném požadavku na změnu počtu řádků resp. při požadavcích na změny ve strukturách řádků.

4.19 Běžná asociace

Další vztah v analytickém modelu tříd, který analytik vyhledává, budeme nazývat „běžná asociace“. Vztah běžné asociace je opět směrový vztah mezi třídami, který vede ke vztahu mezi instancemi. Znamená to, že jeho nalezení udává, v jakém vztahu budou instance, až vzniknou ze tříd.

Vztah běžné asociace je protipólem kompozice: Jedna instance používá druhou instanci, ale tato používaná instance není chápána jako její část. Zatímco synonymem pro kompozici je „majitelství“ (ovládání života mezi instancemi ve smyslu celek-část, např. faktura a její řádky), pro běžnou asociaci je synonymum „zápůjčka“. Instance má sice druhou instanci k dispozici a používá ji, avšak není jejím majitelem. Má pouze „zapůjčen“ ukazatel na její vnější obal. Znamená to, že používaná instance

se rodí a zaniká v kontextu někoho jiného, než je ten, kdo ji v běžné asociaci používá.

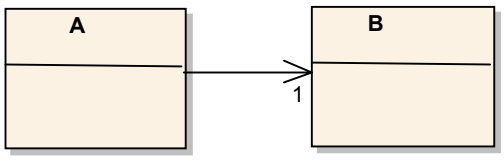
Běžná asociace se vyskytuje ve dvou základních modelových případech a tyto případy (nebo také vzory) jsou pro analytika signálem pro její zavedení:

1. Vzor „Číselníková vazba“: Jedná se o provázání dvou nezávislých instancí, kdy jedna instance používá druhou instanci, ale není zásadně jejím majitelem. Používaná instance je v seznamu mezi ostatními instancemi, byla z něj nějak vybrána (tj. je znám ukazatel na ni) a jiná instance si na ni přes tento ukazatel ukazuje a používá ji.
2. Vzor „Vztah k parentovi v kompozici ku N“: Jedná se o druhý směr vazby v kompozici ku N, a to v tom případě, kdy se vyžaduje, aby vlastněná instance znala svého majitele (například řádek faktury znal ve své vnitřní struktuře svou fakturu, která jej vlastní). Povaha vazby je úplně stejná jako v předešlém bodě („zápůjčka“), ale modelový případ je jiný.

Oba dva modelové případy si nyní podrobně vysvětlíme.

4.20 Běžná asociace jako vzor „Číselníková vazba“

V tomto případě existuje nějaký seznam výskytů (například číselník), ze kterého se vybere jeden výskyt a ten se prováže na aktuální výskyt, který tento výskyt z číselníku potřebuje. Pomocí syntaxe UML se tento vztah maluje takto:



obrázek 43 Vztah běžné asociace podle vzoru "Číselníková vazba"

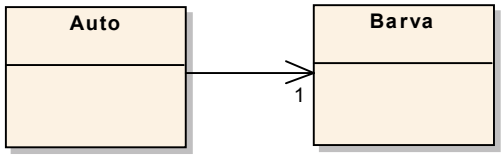
Tímto analytik vyjadřuje tuto myšlenku:

Bude existovat instance ze třídy A, tato instance bude mít k dispozici instanci ze třídy B a může ji používat. Přesto, že ji má k dispozici, tj. „má její ukazatel“ a používá ji, tak tato instance ze třídy B není její část, není to její kompozit. Provázání na instanci

ze třídy B (získání ukazatele na ni) proběhne tak, že instance z B žije samostatně (možná jako kompozit někoho jiného), v rámci nějakého scénáře bude vybrána, tj. identifikována mezi ostatními instancemi ze třídy B, a instance ze třídy A ji dostane jako zápůjčku.

Jako příklad lze uvést informační systém evidence na dopravním inspektorátu. Pro evidenci aut se zavede seznam barev. Existuje a je k dispozici nějaký v dané chvíli konečný seznam barev nazvaný „číselník barev“. Při editaci auta se vybere jedna z existujících barev a ta se prováže s daným výskytem auta.

V modelu tříd se tato vazba zobrazí následujícím způsobem:



obrázek 44 Běžná asociace v použití „číselníkové vazby" ve smyslu „auto má barvu“

Uvedený vztah se interpretuje tak, že výskyt ze třídy auto bude mít k dispozici jeden výskyt ze třídy barva. Výskyt auta může tento výskyt barvy používat, nesmí jej však ani zrušit, ani založit. Pokud se vymaže ze systému daný výskyt auta, potom odpovídající výskyt barvy zůstane v systému zachován. V případě vztahu kompozice je tomu přesně naopak.

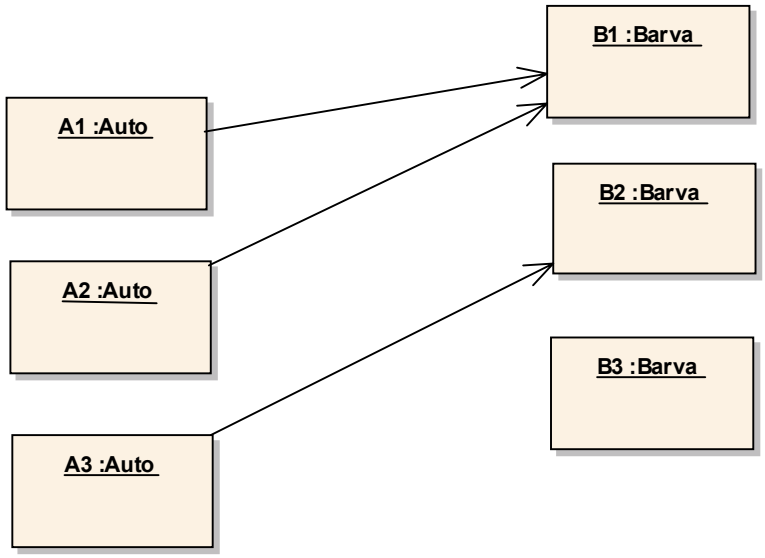
Povaha vazby (běžná asociace versus kompozice) vždy ukazuje, jak se výskyty informací dynamicky chovají. Při scénáři naplnění vazby běžné asociace musí daný výskyt barvy již existovat, poté musí být nějak vybrán ze všech výskytů barvy a následně musí být v dané technologii nějak provázán do výskytu auta. Pokud daný výskyt barvy neexistuje, potom by se ve scénáři výběru muselo odskočit do jiného scénáře, což je nějaký jiný proces nad číselníkem, a tam založit nový výskyt barvy. Teprve poté jej lze provázat v běžné asociaci k autu.

4.21 Základní vlastnost běžné asociace

Základní vlastnost běžné asociace - „číselníkové vazby“ si ukážeme na chování instancí. Pokud totiž analytik zvolí běžnou asociaci, tak na rozdíl od kompozice

vyžaduje, aby pokud má dojít ke shodě informace, tak vyžaduje sdílení instancí. Pro běžnou asociaci je důležitý právě požadavek na shodu ukazatelů, tj. na sdílení instancí.

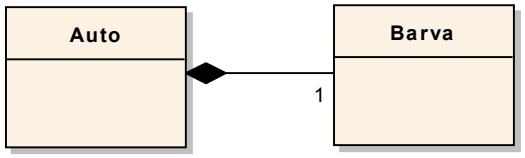
Vezměme si náš příklad s barvami aut. Pokud analytik zvolil běžnou asociaci pro analytickou myšlenku „barva auta“, jak je ukázáno na obrázku viz obrázek 44, tak se vyžaduje následující: Pokud mají dvě instance aut v evidenci tutéž barvu, tak si ukazují a používaly tutéž instanci barvy. V instančním modelu si tuto vlastnost může zobrazit takto:



obrázek 45 Instanční model evidence aut a barev

Na předešlém obrázku je zřetelně vidět, že dvě instance ze třídy Auto označené jako A1 a A2 mají tutéž barvu a to tak, že používají (sdílejí) tutéž instanci B1 ze třídy Barva, zatímco auto A3 má jinou barvu, přesněji řečeno používá jinou instanci ze třídy Barva. Instanci B3 nepoužívá žádná z uvedených tří instancí A1, A2, A3.

Zajímavá situace by nastala, pokud by se nezvolila vazba mezi instancemi ze třídy Auto a instancemi ze třídy Barva jako běžná asociace, ale jako kompozice. Odpovídající analytikův návrh by poté vypadal například takto:



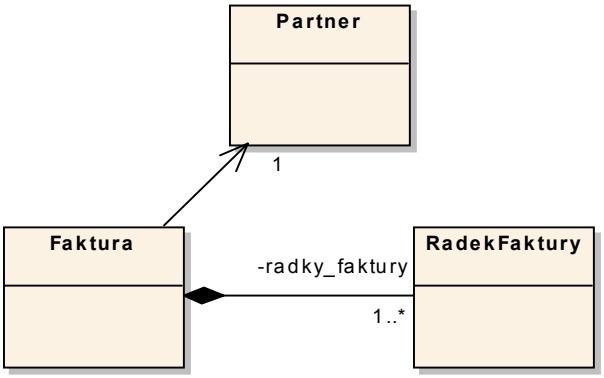
obrázek 46 Barva jako kompozice v autě

V tom případě by tento model odpovídal nežádoucí situaci, kdy s každou novou instancí ze třídy Auto vznikne „jeho vlastní osobní“ nová instance ze třídy Barva bez ohledu na cokoliv. Obsluha by při zadávání nového auta mohla editovat barvu jako vždy nový a nový „vlastní řetězec“, tj. nevybírala by ze seznamu barev, ale doslova by „ťukala“ údaje barvy znovu a znovu pro každé nové auto. Při tomto postupu by nikdo nebyl vázán žádným pravidlem výběru, pouze by barvu znovu zadal jako řetězec. Díky této libovůli by se mohly objevit barvy jako „červeno-červeno-hnědá“ nebo „fialkově-zelená“, jak si obsluha smyslí. Právě proto, aby se zamezilo této nežádoucí kreativitě a získala se jednoznačnost, použije se vztah běžné asociace ve vzoru „Číselníková vazba“.

Samozřejmě je v kompetenci analytika, jaký typ vztahu bude zaveden. Tím analytik určuje chování aplikace. Od volby vztahu se odvíjí další práce designéra, který již díky této povaze vazby takříkajíc nemůže vytvořit jiný vztah. Designér musí pouze tento typ vazby realizovat v konkrétní technologii a k tomu mu pomáhá omezený výčet možných realizací. Mohu jenom zdůraznit, že 90% práce analytika je rozhodování o typu vazby.

Uvedený příklad na barvy aut je názorný. Ukazuje, jak vlastně analytik zjistí, zda se jedná o kompozici nebo o běžnou asociaci: Položí se dotaz: Plní se vazba nějakým výběrem (například obsluhou) anebo se zadává (edituje) napřímo? Barva se vybírá, a to je signál pro zavedení běžné asociace a nikoliv kompozice.

Uvedený vzor „Číselníková vazba“ lze zobecnit pro libovolné informace, které se potřebují pouze provázat a tento vztah nemá přitom povahu kompozice. Například daná faktura má svého partnera (dodavatel - odběratel). Instance partnera musí být faktuře známa a získává se výběrem, tj. není jeho kompozitem. V modelu lze tedy číst následující vztah:



obrázek 47 Běžná asociace na partnera a kompozice řádků faktury

Vztah instancí ze třídy faktura vůči partnerovi je jiné povahy než vztah vůči řádkům faktury. Svou povahou je vztah k instanci ze třídy Partner stejný jako vazba do číselníku. Pro daný výskyt faktury se musí výskyt partnera vybrat z již existujících výskytů. Daná faktura si na něj pouze „ukáže“ (někdy se říká „vidí“), ale neobsahuje jej jako kompozici. Při vymazání faktury partner v systému zůstává. Povahou je tato vazba podle vzoru „číselníková vazba“.

4.22 Vlastnost „isNavigable“

Vazba běžné asociace v obrázku k používané instanci je v modelu označena šipkou. Tato šipka souvisí s vlastností „*isNavigable*“ u konce vztahu, tj. s již zmíněnou směrovostí, někdy se nazývá také jako propustnost. Pokud je konec vztahu označen touto šipkou, znamená to, že ve směru šipky je možnost viditelnosti od jedné instance k druhé, avšak nikoliv naopak. Pokud není v modelu zobrazena šipka, chápe se to jako možnost viditelnosti (propustnosti) v obou směrech.

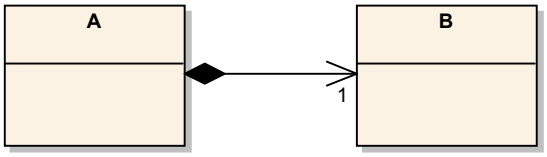
V případě běžné asociace typu „Číselníková vazba“ šipka vyjadřuje tu zřejmou skutečnost, že instance ze třídy Auto má sice k dispozici instanci ze třídy Barva, může ji používat a „ví o ní“, ale obráceně to neplatí. Pokud se podíváme na vnitřní strukturu informace ze třídy Barvy, tak tato informace neví nic o nějakých autech.

V důsledku se tato vlastnost jednostranného vztahu projeví přímo v tom, že naprogramovaná část systému evidující auto potřebuje část systému evidující barvy, tj. třída auto ve své vnitřní struktuře potřebuje třídu barev, a nikoliv naopak. Číselník barev je samostatně stojící entita.

Například při mapování do OOP se tato skutečnost projeví přímo prakticky tak, že třída `CAuto` (ve smyslu OOP) potřebuje třídu `CBarva`, avšak třída `CBarva`

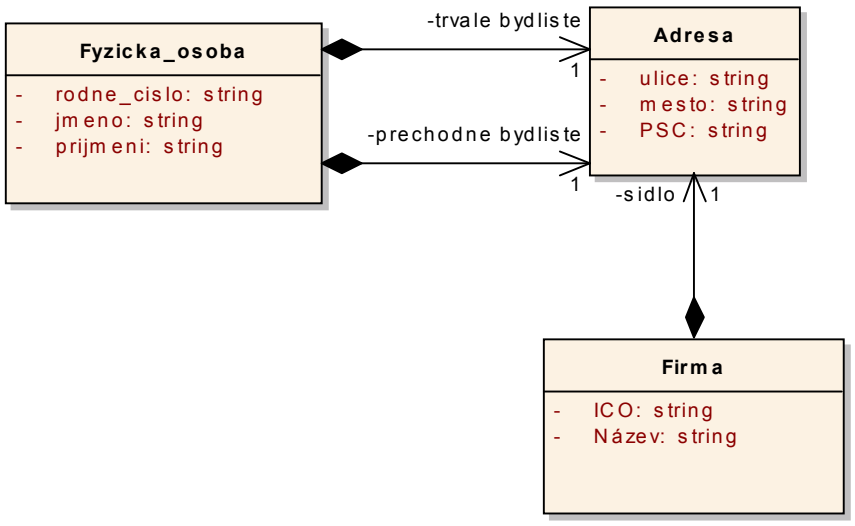
nepotřebuje třídu `CAuto`. Znamená to, že například třída `CBarva` může být implementována v komponentě, kterou si komponenta s třídou `CAuto` přilinkuje.

Vraťme se ještě ke vztahu kompozice ku jedné, nyní již při znalosti vlastnosti „isNavigable“. Kompozici ku jedné jsme si demonstrovali na řešení adres, viz *obrázek 21*. Pokud se zamyslíme nad tím, kdo koho v tomto vztahu vlastně používá, kdo koho musí a nesmí „vidět“, tak je zřejmé, že instance ze třídy Adresa jsou jako „vložené“ a svými majiteli ovládané instance viděny svými majiteli, ale naopak, tyto instance nevědí „kým“ jsou používány a nevidí svého majitele. Správněji by tedy měl být v kompozici ku jedné preferováno použití vlastnosti „isNavigable“ ve směru šipky takto:



obrázek 48 Kompozice ku 1 je směrová

a u našeho příkladu s adresami by měl být model změněn na relevantní vztah takto:



obrázek 49 Model s kompozicí ku jedné upravený pomocí směrovosti vazby

V předešlých kapitolách jsme tuto syntaxi nepoužívali z důvodu posloupnosti a logiky výkladu.

4.23 Kvalifikace vazby

Pokud je třeba pracovat s auty pouze určité barvy, není to problém barvy jako takové. Musí se vybrat ze všech výskytů ta auta, která tuto barvu mají. Tento dotaz na auta je však problém aut a nikoliv problém barvy. Analytik tuto skutečnost „filtru“ vyjádří na úrovni výskytů aut. Poté se toto mapuje do designu. Například v technologii s relační databází se může pro takový filtr nad vazbou v designu využít nějaký SELECT nad tabulkou aut apod.

V souvislosti s touto problematikou se zavádí jeden důležitý pojem, který se nazývá kvalifikace vazby.

Nechť instance ze třídy A používá instanci ze třídy B ve smyslu „číselníkové vazby“, tj. v modelu tříd se objevila odpovídající vazba běžné asociace mezi třídami A a B (viz *obrázek 44*).

Kvalifikace vazby znamená nasazení výběrové podmínky na množinu instancí ze třídy A, která nějak omezuje tuto množinu vzhledem k této vazbě. Velmi často je potřebná výběrová podmínka vyjádřená formulací „všechny instance ze třídy A, které mají vztah k jedné instanci B“. Například v případě evidence aut se jedná o podmínku „všechna auta jedné této dané barvy“. Nasazení takové podmínky na vazbu budeme nazývat kvalifikace vazby.

Analytik si může představit, že kvalifikace vazby probíhá tak, že se sekvenčně projdou všechny instance ze třídy A, posoudí se podmínka pro danou instanci a určí se výsledek „platí - neplatí“. Designér uvedenou podmínku může vyřešit například SQL filtrem nad relacemi v RDB.

4.24 Běžná asociace podle vzoru „Vztah k parentovi v kompozici ku N“

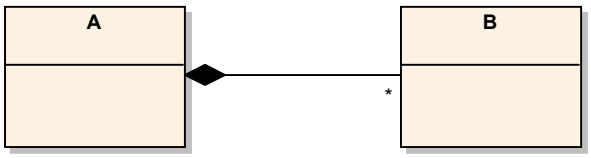
Další situace, která vede k použití běžné asociace, je vzor pro zpětný (obrácený) vztah v kompozici jedna ku N, v programátorské hantýrce se nazývá „vztah k parentovi“. Jako příklad lze uvést fakturu a její řádky, viz *obrázek 24*. V tomto případě vztah k parentovi ukazuje, v jakém vztahu je instance řádku faktury vůči svému majiteli, zde k instanci faktury. Instance řádku faktury by měla znát svou instanci faktury, protože pokud se pracuje s instancí řádku, potřebuje se používat odpovídající instance faktury.

Ukazuje se, že tento „vztah k parentovi“ má úplně stejnou povahu a vlastnosti, jako má již uvedená „číselníková vazba“ a také se úplně stejně mapuje do technologie

designu. Daná instance řádku faktury dostane již existující instanci svého majitele k dispozici přes ukazatel jako zápůjčku a prováže se na ni, tj. ukáže si na ni. Výsledkem je, že instance řádku faktury „vidí“ svého majitele (tj. parenta) instanci faktury stejně, jako když nějaká instance „vidí“ prvek z číselníku v předešlé kapitole. Jinak řečeno řádek faktury má technologicky stejnou vazbu na svou instanci faktury, jako když auto „vidí“ svou barvu z číselníku barev. Všimněme si, že všechny řádky z téže faktury „vidí“ stejnou instanci svého majitele instance faktury, tj. majitel se sdílí. Dá se říci, že instance faktury drží své instance řádků jako kompozici a obráceně instance řádku „vidí“ svou instanci faktury jako parenta v běžné asociaci. Vazba na parenta se naplní v okamžiku zrodu instance řádku faktury a v životě instance řádku se již nemění, tj. instance řádek neputuje nikam k jinému majiteli.

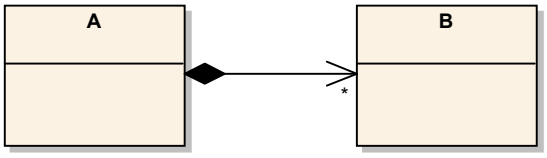
Je otázkou, jak v UML znázornit předešlé úvahy. Běžná asociace se jako vazba na parenta ve vztahu kompozice ku N znázorňuje v UML v CLASS MODELU pomocí tzv. směrovosti vazby kompozice, tj. pomocí již zavedené vlastnosti „isNavigable“. Důležité je, zda se při kompozici ku N použije nebo nepoužije šipka udávající tuto vlastnost směrovosti „isNavigable“.

Pokud se nepoužije šipka, tj. diagram se namaluje takto:



obrázek 50 Kompozice spolu s běžnou asociaci zpět

potom autor vyjádřil tu skutečnost, že vztah od B k A je běžnou asociací a tedy instance z B vidí svého parenta a potřebuje jej. Pokud se naopak v UML použije směrovost vazby s šipkou takto:



obrázek 51 Jednosměrná kompozice ku N

potom instance ze třídy B „nevidí“ instanci ze třídy A, tj. instance ze třídy B o instanci ze třídy A (o svém parentovi) nic neví.

Poznámka: Některé CASE nástroje tuto vlastnost zavádějí analogicky jako tzv. Direction daného vztahu. Pokud se zvolí u Direction odpovídající směr (např. SOURCE-DESTINATION), objeví se u vztahu šipka, v případě obrácení směru Direction se objeví obrácená šipka.

Situaci lze přirovnat k možnosti „skočit“ nahoru v adresářové struktuře. Pokud zvolíme model podle obrázku *obrázek 50 Kompozice spolu s běžnou asociaci zpět*, dáváme tím jako analytici najevo podobný požadavek, jako by ten, kdo bude v určitém adresáři, mohl „skočit“ na nadřízený vyšší adresář. Pokud zvolíme model podle obrázku viz obrázek 51, dáváme tím najevo, že daný adresář nezná „cestu nahoru“, tj. pokud někdo drží instanci kompozitu, nezná se od něj jeho majitel.

Ve většině případech se u evidenčních informačních systémů vyžaduje, aby v kompozici ku N existoval i obrácený vztah běžné asociace ve směru na parenta (tj. „možný skok nahoru“), proto se většinou používá model podle obrázku viz *obrázek* 50 Kompozice spolu s běžnou asociaci zpět*,* a nikoliv podle obrázku viz *obrázek 51*.

Existují však případy, kdy opačný směr vazby na parenta je buď přímo zakázán anebo se nevyžaduje, aby existoval, tj. vyžaduje se, aby platil *obrázek 51*. V praxi jsem setkal jsem s dvěma základními modelovými případy, kdy tato situace může nastat.

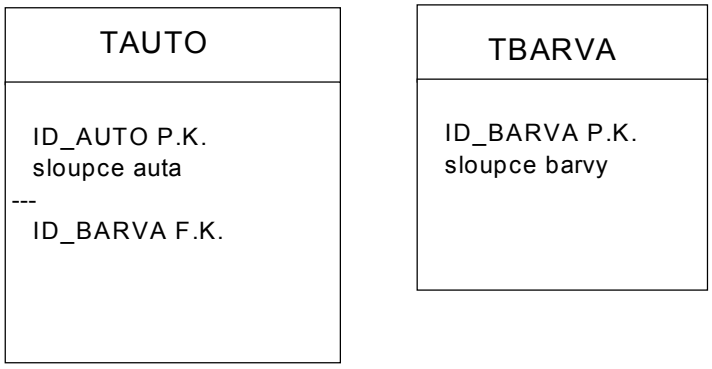
- První ze situací, kdy se nevyžaduje vztah k parentovi, se vyskytuje u technologických systémů (nikoliv evidenčních), kdy jednotlivé kompozity komunikují mezi sebou pouze „shora dolů a nikdy jinak“. Představme si například řešení nějakého stromu komponent, které dostávají signály pouze shora a rozposílají je dolů svým částem - kompozitům dolů. V tom případě v každém z těchto kompozitů dochází pouze k přeposlání zpráv dolů k dalším svým částem a nikdy se neposílá žádný požadavek vzhůru k parentovi. V tomto případě je vazba na parenta zbytečná.
- Jiný a velmi častý případ souvisí s vazbou, kterou jsme zatím neprobrali, takže ji uvedeme nyní pouze krátce. Může se stát, že daná instance ze třídy B na obrázku viz *obrázek 51* „nevidí“ svého parenta proto, protože tuto vlastnost má až tzv. dědic této třídy ve vztahu GENERALISATION-SPECIALISATION (tj. dědic ve vztahu dědičnosti). Tento případ můžeme podrobně vysvětlit až probereme zmíněný vztah GENERALISATION-SPECIALISATION (dědění).

Pro vztah k parentovi bývá pro vývojáře relačních databází matoucí jedna důležitá okolnost: V relační databázi přímo mezi daty je vždy v datech v kompozitu ku N znám parent, protože pokud držíme záznam kompozitu, tento záznam v sobě obsahuje cizí klíč parenta. Tento jev souvisí s povahou relací v RDB, která není sama o sobě směrová a je vždy symetrická. V aplikaci nad touto databází však může

nastat situace, kdy se tato vazba v tomto směru (od kompozitu k parentovi) nikdy nepoužije.

4.25 Mapování běžné asociace do RDB

Mapování do RDB začněme u vzoru „Číselníková vazba“ a vyjděme z modelu na obrázku viz obrázek 44. Při návrhu RDB se v tomto případě používá pouze jeden způsob mapování, kdy se vytvoří jak tabulka pro auto TAUTO, tak tabulka pro barvy TBARVA a cizí klíč putuje z barvy do auta, např. takto:



obrázek 52 Mapování číselníkové vazby do RDB

Všimněme si, že mapování je ve strukturách tabulek stejné, jako je u kompozice ku jedné (viz příklad s adresami). Rozdíl je však v chování instancí. Dvě auta se stejnou barvou budou mít na předešlém obrázku stejnou hodnotu cizího klíče ID_BARVA, čímž si v relační databázi ukážou na tentýž záznam v tabulce TBARVA. To při kompozici nikdy nenastane, protože při kompozici ku jedné má každý záznam v jedné tabulce svůj „osobní“ vlastní vedlejší záznam v kompozici.

Mapování vztahu k parentovi v RDB si nemusíme zvlášť ukazovat: Stačí využít již existujícího mapování podle obrázku viz obrázek 40. Podle tohoto mapování je zřejmé, že každý kompozit v seznamu (záznamy v tabulce pro řádky faktury) obsahují cizí klíč na záznam v tabulce pro faktury, tedy na záznam parenta. Tohoto klíče lze jednoduše využít i v aplikační úrovni, pokud se to vyžaduje.

4.26 Příklad na adresy - pokračování

Dlouhou dobu jsem se domníval, že učebnicovým příkladem na vztah kompozice ku jedné jsou právě adresy řešené podle obrázku viz obrázek 21. To jsem si myslel až do chvíle, než jsem přednášel v jedné pobočce firmy v Hradci Králové. Tato firma mimo jiné řeší rozsáhlou agendu adres občanů pro státní správu.

Jeden z posluchačů se při školení ozval: „Jestli tomu, co nám přednášíte, dobře rozumím, tak my vyvíjíme adresy, ale nikoliv jako kompozici ku jedné, ale jako běžnou asociaci."

Můj následný dotaz zněl sice trochu slangově , ale výstižně: „Chcete tím říci, že vedete číselník všech adres?"

„Ano, přesně tak."

Otázka nyní zní: Jaký by nastal základní rozdíl, pokud bychom adresy vyřešili jako běžnou asociaci a nikoliv jako kompozici ku jedné? Odpověď zní: Rozdíl je podstatný. Spočívá v tom, že při běžné asociaci se instance adres budou v nějakém seznamu, poté se „nějak" vybírat z tohoto seznamu a následně budou používány jako zápůjčky těm instancím, které tyto adresy používají. Adresy se nebudou rodit v kontextu svých uživatelů, ale budou žít jako samostatný nezávislý seznam, do kterého se bude ukazovat „to je moje adresa". Například pokud dvě instance fyzické osoby budou mít v evidenci stejnou adresu, znamená to skutečně „stejnou instanci adresy" a po mapování do relační databáze toto sdílení instance adresy povede shodě hodnot použitého cizího klíče adresy ID_ADRESA ve dvou záznamech.

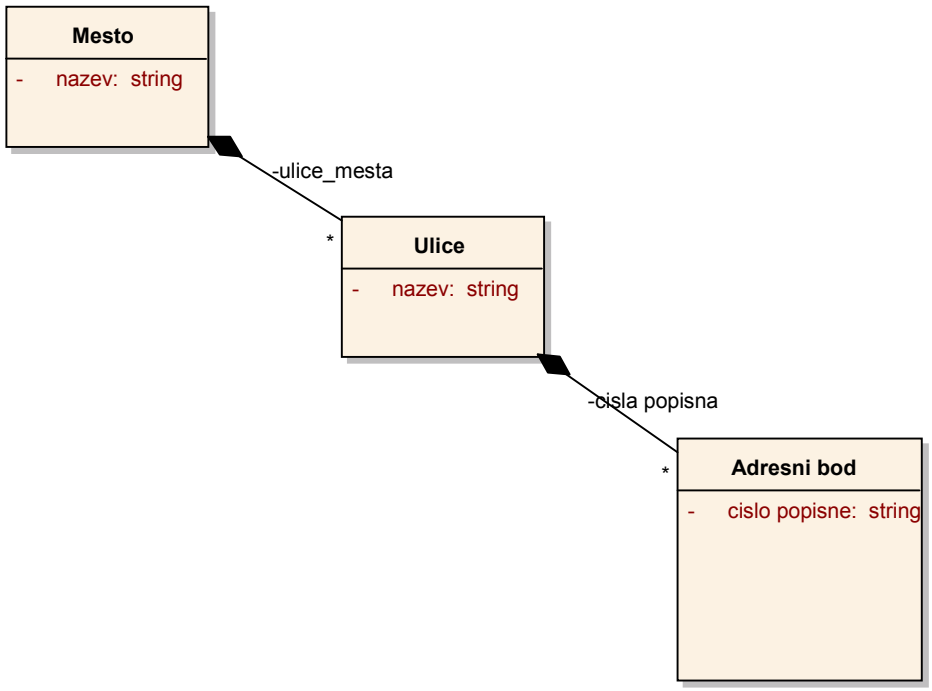
Pokusme se rozvinout tuto myšlenku trochu dále a porovnejme obě řešení. Samozřejmě protože se jedná o příklad, řešení bude velmi jednoduché a spíše sloužící jako ukázka, jak se myslí ve fázi analytického modelování a následně v designu.

Adresu, která bude řešena pomocí běžné asociace, budeme raději nazývat Adresní bod, abychom tato dvě řešení odlišili. Pokud se má Adresní bod vybírat ze seznamu, potom by bylo vhodné zvolit konstrukci postupného výběru, například takto: Můžeme si představit, že obsluze se při výběru adresního bodu nejprve zobrazí seznam měst, po výběru města se zobrazí seznam ulic tohoto města, po vybrání ulice se zobrazí seznam povolených čísel popisných této ulice.

Nyní se naskýtá tato otázka: Pokud zvolíme toto řešení, tak v jakém vztahu je ulice a město? Je to kompozice anebo ne? Základním kritériem rozhodnutí je v tomto případě možnost sdílení instancí. Na první pohled by se mohlo zdát, že sdílení ulic je možné, vždyť například v Praze je ulice Milady Horákové a v Brně je také ulice

Milady Horákové. Ale v této větě je skryt malý chyták: V každém případě se jedná o dvě různé ulice, například pokud se jedna přejmenuje, druhá to „nepocítí“. Tyto dvě ulice jsou dvě různé instance evidovaných ulic, u nichž došlo ke shodě v hodnotách názvů.

Nejjednodušší řešení měst, ulic a čísel popisných může být následující (upozorněme, že mohou existovat i další řešení, které nyní nemůžeme probrat, protože nemáme k dispozici veškerou nutnou syntaxi):

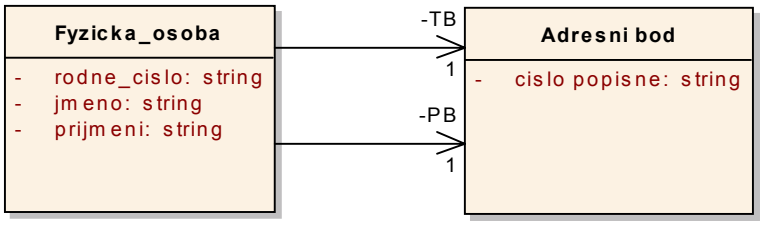


obrázek 53 Jednoduchá evidence adresních bodů jako čísla popisná v evidované ulici v evidovaném městě

Všimněme si, že u kompozice není šipka, tj. uvedené propojení mezi třídami se chápe jako dva směrové vztahy „tam i zpět“: Na jedné straně město „drží“ svoje ulice. Znamená to, že ten, kdo jako klient drží instanci města, může toto město požádat o něco, co se týče ulic tohoto města, tj. co se opře o tuto vnitřní strukturu města s ulicemi. Na straně druhé každá ulice má ve své vnitřní struktuře odkaz na instanci města, do kterého patří (ve vztahu není šipka „isNavigable“). Pokud klient drží danou instanci ulice, může tuto ulici požádat o něco, co se týče jejího města, protože ho zná. Úplně stejně je tomu ve vztahu ulice versus adresní bod. Kdo drží instanci ulice, může ji požádat o něco, co se týče jejích adresních bodů a naopak, pokud někdo drží adresní bod, může jej požádat o něco, co se týče jeho ulice, kam patří.

Pokud dáme úvahy předešlého odstavce dohromady, potom si dovedeme představit, jak analytik uvažuje nad touto otázkou: Nějaký klient (například instance fyzické osoby) si ukazuje na adresní bod. Můžeme vypsat na obrazovku název města, název ulice a číslo popisné této adresy? Odpověď zní ano, stačí jenom poskládat uvedené možnosti požadavků mezi sebou.

Fyzická osoba, která bude potřebovat adresu trvalého bydliště a adresu přechodného bydliště, se prováže běžnou asociací dvakrát na adresní bod definovaný v předešlém diagramu, například takto:



obrázek 54 Adresy (adresní body) použité pro fyzickou osobu v běžné asociaci

Všimněme si, že jsme nenamalovali předešlý celý diagram s městy, ulicemi a adresními body, tj. obrázek 53, znovu. Díky uvedeným rekurzivním vlastnostem „kdo koho může použít“ se nemusíme opakovat a stačí namalovat vztah pouze mezi fyzickou osobou a adresním bodem. Vše další vyplývá z již předtím uvedených diagramů.

Tato vlastnost se může jevit jako velmi nepříjemná, protože na daném obrázku „nevidíme všechno“ (vzpomeňme na Českou Třebovou ☺). Avšak právě tato možnost dekomponovat myšlenky je nejsilnější zbraní analytika, která mu umožňuje rozložit složité myšlenky na několik jednoduchých.

Nyní má analytik před sebou dvě možná řešení pro agendu adres: Jedno s kompozicí a druhé s běžnou asociací. Musí si vybrat: Jedno z nich použije pro evidenci adres v systému. Mohli bychom si nyní položit tuto otázku: „Jak to vlastně je? Které z těchto dvou řešení je to správné?“

Je třeba si uvědomit, že takto položená otázka není relevantní. Neexistuje „správné“ nebo „nesprávné“ analytické řešení nebo neplatí rozhodování ve smyslu „tak to je a tak to není“. Analytik je vývojář, který navrhuje informační systém, a je tedy tvůrcem toho, co se žádá. Snaží se o analytický návrh systému, který bude lepší, užitkovější, případně méně pracný apod. Tedy obě řešení mohou být za různých okolností tím „co se žádá“ a každé z nich může být považováno za jiných okolností za „to lepší“. Analytik vybírá mezi řešeními nikoliv podle toho „jak to je“, ale podle toho, které z těchto řešení přinese více výhod a které méně nevýhod.

Zkusme si vyjmenovat všechny výhody a nevýhody, které přinášejí obě řešení adres. V tomto případě většinou výhody jednoho řešení jsou nevýhodami druhého a naopak. V následujícím odstavci jsou tyto výhody a nevýhody vyjmenovány tak, jak je určovali posluchači na školeních:

1. Nevýhodou řešení adres s kompozicí je redundace adres, tj. opakování adres. Může se to projevit například při exportech a importech apod.
2. Nevýhodou řešení adres s kompozicí je jejich nejednoznačnost. Adresy se zadávají různě, někdo napíše „náměstí T.G.M.“, někdo napíše „nám. T.G.M“, jiný zase „náměstí T.G.Masaryka“ a máme tři různé adresy. Řešení s běžnou asociací sice nevyloučí chybu (když obsluha chybně vybere danou instanci), ale v řešení s kompozicí není vůbec žádný nástroj, který by zaručil požadovanou jednoznačnost. Některé systémy vyžadují jednoznačné a přesné adresy bezpodmínečně kvůli požadovaným výstupům, např. určeným pro státní úřady (např. výpisy pro finanční úřady, výpisy pro ČNB apod.). V těchto systémech by se mělo nejednoznačnostem adres v každém případě zabránit.
3. Nevýhodou adres v kompozici je problém, který nastane při přejmenování měst a ulic. U běžné asociace lze tento problém vyřešit přímo u dané jedné instance města nebo ulice, v kompozici se musí všechny stejné adresy vyhledat a všude zaznamenat nový historický záznam. Spolu s bodem 2 se může jednat o takřka neřešitelný problém vedoucí až k hnusnému ručnímu zpracování.
4. Nevýhodou řešení s běžnou asociací je možnost mít „příliš mnoho zbytečných adres“ v případě, kdy by existoval obrovský seznam adres a málo se jich používalo.
5. Současně s předešlým bodem nastává otázka, jak řešit případ adresního bodu, který v systému není. Část aplikace musí být řešena jako správa adres se vším všudy, tj. agenda adres musí být k novým adresám učenlivá.
6. Výhodou řešení s kompozicí je její jednoduchost hraničící až s primitivností implementace.

Když posoudíme všechny vyjmenované výhody a nevýhody (možná vás napadnou i další), tak můžeme dojít k následujícímu závěru:

Pokud bychom navrhovali systém, kde adresy hrají roli pouze kontaktu, například si představme zásilkovou službu, knižní zásilkový klub apod., tak potom si vystačíme s řešením, které odpovídá kompozici ku jedné. Adresy se v tomto případě chovají pouze jako přívěsky „shluků tří řetězců město, ulice PSČ“ a nic víc. Zákazník takového zásilkového systému se stará o svou adresu sám. Pokud se přestěhuje anebo mu přejmenují ulici, nahlásí tuto skutečnost například mailem. Pokud ovšem

někdo bude potřebovat nějaké marketingové přehledy (a kolik klientů máme z Pardubic?) hrozí reálné riziko, že nedostane relevantní výsledky.

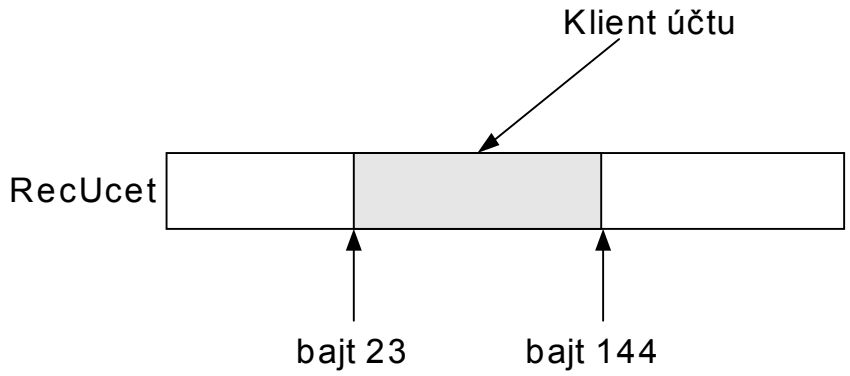
Pokud by se jednalo o bankovní systém resp. jiný podobný systém, který dává různé výkazy pro úřady včetně statistik, tak adresa by nemohla být řešena takto jednoduše. V těchto případech bychom se zřejmě přiklonili k běžné asociaci.

4.27 Příklad na chybné určení kompozice a běžné asociace

Jak vidno na předešlém příkladu, jedním ze základních úkolů analytika je určovat povahu vazbu mezi instancemi v modelu tříd, tj. zatím v této fázi výkladu mezi kompozicí a běžnou asociací. Pokud se analytik zmýlí a vazbu určí chybně, tak se tato chyba promítne do chování systému velmi nepříznivě přímo s fatálními důsledky. Jako příklad takovéto fatální chyby si uvedeme následující chybnou konstrukci zavedenou již v analytickém modelu.

Jeden bankovní informační systém byl řešen souborově v syntaxi Pascalu, vyskytovaly se v něm pro práci se soubory záznamy typu record. Při vývoji byla podstatně zanedbána dokumentace analytických modelů a většinou se navrhoval buď rovnou design anebo se šlo dokonce rovnou do kódu bez dokumentace designu. Důsledkem tohoto postupu při neznalosti základních vztahů analytického modelu tříd se v kódu objevilo následující řešení, které vedlo následně k obrovským problémům:

V systému se vyskytovaly bankovní účty, což není v bankovním systému nic překvapivého. Tyto účty byly realizovány pomocí záznamu record, označme jej například jako **`RecUcet`**. Každý účet má svého majitele, tj. klienta, kterým může být buď fyzická nebo právnická osoba. Tento analytický požadavek se vyřešil tak, že v daném recordu **`RecUcet`** se jeho určitá část (jako příklad od bajtu 23 po bajt 144) vyhradila pro údaje klienta. Nazvěme tuto část například jako Klient účtu. Do této části recordu se zapisovaly údaje klienta: Pokud se jednalo o fyzickou osobu, tak se na určitá místa bajtů umístily do části Klienta účtu informace jako rodné číslo, jméno, příjmení, adresa atd., pokud se jednalo o právnickou osobu, tak se na totéž místo umístily údaje jako IČO, název firmy, adresa sídla. Graficky bychom mohli tuto situaci znázornit takto:

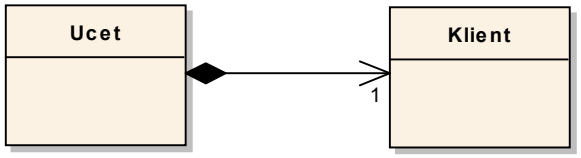


obrázek 55 Nepříliš šťastné řešení klientů účtů v recordu Pascalu

Tušíte již nyní, k jakým neskonalým problémům díky tomuto řešení docházelo? S každým novým a dalším účtem se musely povinně údaje klientů opakovat. Výsledkem byl z hlediska evidence klientů totálně nepřehledný systém.

Pokusme se tento problém řešit při našich znalostech analytického modelování a konkrétně pomocí analytického modelu tříd se pokusíme vyjádřit analyticky „profesionálně“:

Základní chybou předešlého modelu je ta skutečnost, že byla na analytické úrovni (nevědomky) zvolena vazba mezi účtem a klientem jako kompozice:

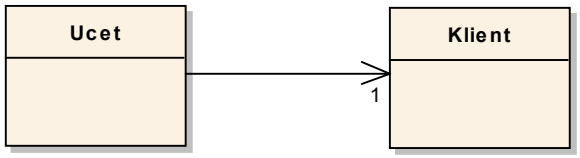


obrázek 56 Analytický model předešlého řešení v recordech

Všimněme si, že chyba není vůbec v tom, že by se problém řešil pomocí recordů v souborech. Základní chyba je již v analytickém modelu, který pochopitelně díky metodě TUNEL v dané firmě vůbec v dokumentaci neexistoval (mimochodem, s největší pravděpodobností, pokud by existoval, tak by nesmysl na předešlém obrázku určitě někoho „trknul“).

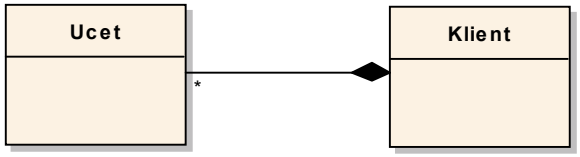
Důsledky nesprávného řešení s kompozicí jsou opravdu velmi nepříjemné, protože klienti se v účtech opakují. Identifikovat shodu u chybně zadaných údajů klienta je velmi obtížné. A to nehovoříme o přehledech přes klienty, které se programovaly nepředstavitelně složitě.

Správné řešení samozřejmě spočívá v opuštění kompozice a v použití běžné asociace. Ale zde se vyskytne druhý malý háček. Je sice zřejmé, že účet má mít ke klientovi vztah běžné asociace, ale otázka zní: Jedná se o běžnou aosciaci z číselníkové vazby anebo se jedná o zpětný vztah k parentovi? Existují totiž dvě možnosti, které se analytikovi nabízejí. Buď se rozhodneme pro číselníkovou vazbu (podle vzoru Auto vidí svou Barvu):



obrázek 57 Řešení pomocí číselníkové vazby

anebo se rozhodneme řešit problém pomocí obrácené kompozice ku N (podle vzoru „Řádek faktury vidí svou Fakturu“):



obrázek 58 Podobné řešení pomocí kompozice ku N

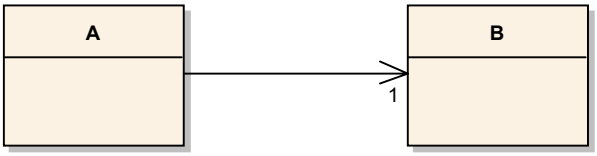
V obou případech je totiž vztah od účtu ke klientovi žádoucím vztahem běžné asociace, ale jedná se o dvě možná řešení.

Příklad ukazuje, že analytik musí při použití běžné asociace vyřešit otázku, který z těchto dvou možných vztahů vybrat, zda číselníkovou vazbu anebo vztah k parentovi. Tomuto problému je věnována následující kapitola.

4.28 Běžná asociace jako číselníková vazba anebo jako zpětná vazba na parenta?

Porovnejme oba předešlé obrázky a vysvětleme si, které z těchto dvou řešení kdy a za jakých okolností vybereme.

Každý model vytvořený analytikem ve fázi analytického modelování má povahu zadání pro další fáze vývoje, tj. pro design a kódování. Ukažme si tedy, jaké zadání pro designéra vysloví analytik při modelu podle vzoru číselníkové vazby v tomto případě:



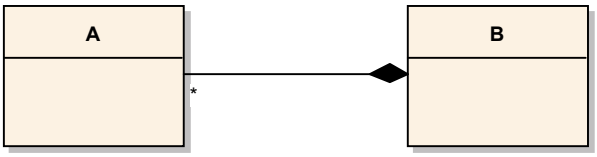
obrázek 59 Číselníková vazba A "vidí" B je zadáním pro designéra o viditelnosti

V tomto případě analytik na jedné straně vyslovuje myšlenku, že instance ze třídy A bude používat jednu instanci ze třídy B jako zápůjčku, na straně druhé však současně říká: Z druhé strany, viděno od instance B (tj. v pohledu její vnitřní struktury) instance ze třídy B neví vůbec nic o nějaké třídě A a nesmí nic vědět! Nedej bože, designére, aby jsi systém navrhl tak, že by třída B a její instance potřebovaly pro svůj život cokoliv od A. Programátorsky řečeno, části kódu, které budou realizovat život instancí ze třídy B, nebudou mít v sobě žádnou zmínku, žádný typ, žádné volání, žádné linkování atd., které by se jakýmkoliv způsobem týkaly částí kódu obsluhující život instancí ze třídy A. V tomto smyslu třída B neví nic o existenci třídy A a pokud by třída A byla ze systému vyjmuta, tak třída B „nezařve“, že jí něco chybí a „žije klidně dále“. Třída B v tomto smyslu žije „svým vlastním životem“.

V designu v pohledu fyzického návrhu aplikace do komponent je zřetelně vidět, že přesně napříč touto vazbou by bylo možné provést střih systému do dvou komponent, kde jedna komponenta bude obsahovat třídu A a druhá komponenta bude obsahovat třídu B, přičemž komponenta obsahující třídu A si musí linkovat komponentu obsahující třídu B.

Připomeňme si na v modelu velmi často používanou podmínku, která umožňuje vybrat všechny instance ze třídy A, které si ukazují na tutéž instanci B. Takovouto podmínku jsme nazvali kvalifikací vazby. Například u aut se jedná o podmínku „všechna auta této barvy“. Při realizaci této podmínky neprogramujeme část systému obsahující třídu Barva, ale část systému obsahující seznam instancí aut. Tato podmínka se aplikuje na seznam aut, přičemž daná instance barvy je vstupním parametrem této podmínky. Není to tedy tak, že pokud chceme „všechna auta této barvy“, že bychom, se ptali dané instance barvy. Je to tak, že chceme-li v této číselníkové vazbě „všechna auta této barvy“, tak se ptáme seznamu aut, kterému (tj. seznamu) tuto barvu podvrhneme jako vstupní parametr tohoto dotazu. Barva o „svých“ autech nic neví, protože žádná nevlastní.

Všimněme si, že oproti tomu pokud analytik navrhne systém takto:



obrázek 60 Při vztahu na parenta opět instance z A "vidí" instanci z B

tak z hlediska směrového vztahu od instance ze třídy A k instanci ze třídy B je situace stejná jako v předešlém případě: Opět se jedná o vztah zápůjčky ukazatele na instanci (instance ze třídy A „vidí“ instanci ze třídy B technologicky stejně jako v číselníkové vazbě). Který z těchto dvou vztahů zvolit?

K zodpovězení na tuto otázku je dobré si uvědomit, v tomto případě je situace z hlediska zadání pro designéra diametrálně odlišná v opačném pohledu interakce, tj. v pohledu vnitřní struktury instance ze třídy B. Tato instance vlastní a ovládá své instance ze třídy A, tj. má je jako vlastník ve své moci, může je o cokoliv žádat, nechat rodit a zanikat apod., protože jsou to její „děti“, je to seznam jejích „dětí“. Samozřejmě v důsledku z toho plyne to, že na rozdíl od předešlého případu třída B potřebuje ke svému životu třídu A. Třída A reprezentuje její „části“, které nelze ze systému odebrat. V tomto případě také nelze provést střih do komponent mezi těmito třídami.

Všimněme si, že na rozdíl od předešlého případu s klasifikací vazby je nyní problém „dej svoje části“ již problémem dané instance majitele, která tyto děti ovládá. Rozdíl mezi kvalifikací číselníkové vazby a ovládání dětí v kompozici si můžeme demonstrovat na této představě OOP kódu: Nechť třída B vlastní kolekci instancí ze třídy A. Tomu v představě analytika odpovídá předešlý obrázek s kompozicí. Potom bychom mohli zavést například operaci **`GetChilds(i)`**, která vrací i-té dítě, například takto:

…

```
MyA = MyB.GetChilds(i);
```

…

a instance **MyA** zná svého majitele, kterým je v tomto případě instance **MyB**

Pokud bychom však zvolili běžnou asociaci jako číselníkovou vazbu, tj. vztah podle obrázku viz obrázek 59, tak předešlá konstrukce `GetChilds` je pro instanci **MyB** zakázaná, protože v kódu třídy B se nesmí vyskytovat nic o třídě A, jedná se o „čistý seznam“ neznající, kdo používá jeho prvky („čistý číselník“). Pokud chceme dostat instance, které „vidí“ tutéž instanci, musíme správce tohoto seznamu, například jej nazvěme jako objekt **CollectionOfA**, požádat o kvalifikaci vazby, například takto:

…

```
CollectionOfA.SetFilterB(MyB);
loop i:
 MyA = CollectionOfA.GetItem(i);
```

…

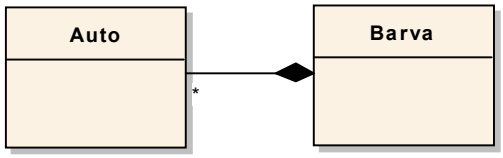
anebo v jedné metodě rovnou takto:

…

```
loop i:
 MyA = collA.GetItemFilteredByB(MyB,i);
```

…

Pokud se analytik rozhoduje mezi těmito dvěma vztahy, pak právě předešlá úvaha, zda má instance ze třídy B znát své děti anebo se jedná o čistou entitu bez těchto dětí (čistý číselník) má rozhodující význam. Hrubou chybou analytického návrhu je, pokud se čistá entita „zašpiní dětmi“, které nejsou její. V našich příkladech by bylo bezesporu hrubou chybou zvolit vztah kompozice u barev takto:



obrázek 61 Hrubá chyba "znečištění" číselníku

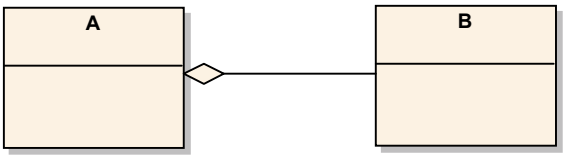
Třída Barva nesmí mít v dosahu své viditelnosti třídu Auto. Třída Barva není na třídě aut závislá, tj. z hlediska syntaxe UML není vůči ní ve vztahu DEPENDENCY (vztah bude probrán dále v druhém díle).

Podobně můžeme rozhodovat ohledně klienta a jeho účtů, viz obrázek 57 a následný obrázek 58: Pokud by namísto klienta stály rovnou osoby jako právnická osoba, fyzická osoba apod., pak by bylo chybou znečistit tyto osoby tím, že vlastní účty. Osoby účty nevlastní a stojí samostatně jako čistý seznam pouze používaný.

Pokud bychom zvolili konstrukci, že existuje entita klient a teprve klient si ukazuje do čistých osob, potom si můžeme dovolit dát klientovi v kompozici účty. Tento příklad bude ještě blíže rozebrán později, protože musíme probrat ještě některé další prvky modelování tříd.

4.29 Sdílená neboli slabá agregace

Vztah sdílené neboli slabé agregace (shared aggregation) se značí takto:



obrázek 62 Sdílená agregace

Vztah je podobný jako kompozice v tom smyslu, že se opět jedná o vztah mezi instancemi a vztah celek versus část, kde kosočtverec označuje celek. Na rozdíl od kompozice však není tento vztah silný. Již neexistuje jednoznačné majitelství instance ze třídy A vůči instanci ze třídy B a mohou se vyskytovat i jiní majitelé instance ze třídy B anebo instance ze třídy B může dokonce žít „samostatně“ (majitelem je systém jako celek).

Důsledkem toho je, že mohou existovat takové scénáře, ve kterých se sice s instancí ze třídy A něco děje (například vymazání), ale instance ze třídy B nemusí v tomto scénáři toto pocítit. Na druhou stranu existují scénáře, kdy se daná instance ze třídy A chová vůči instanci ze třídy B jako majitel stejně jako v kompozici.

Zjednodušeně řečeno vztah sdílené agregace implikuje chování mezi instancemi „v některém scénáři jako kompozice a v některém scénáři nikoliv“.

Dá se z toho usoudit, že rozdíl sdílené agregace oproti kompozici spočívá ve větší volnosti majitelství. Instance ze třídy A v některých scénářích působí jako majitel, v některých scénářích nikoliv (majitelem je někdo jiný). Na rozdíl od toho v kompozici je identifikace vnitřního prvku dána kontextem jeho majitele a to vždy, bez výjimky a nikdy jinak. Ve sdílené neboli slabě agregaci toto pravidlo jednoznačnosti neplatí.

Z praktického hlediska se rozdíl mezi kompozicí a sdílenou agregací v určité fázi analytických prací příliš nerozlišuje, protože nemohou být známy všechny možné scénáře a tudíž všechna možná majitelství. Tento vztah je upřesňován později. Nutno ale podotknout, že kompozice bývá identifikována z povahy věci většinou mnohem dříve než sdílená agregace. Například je zřejmé, že řádek faktury jednou vzniklý v dané instanci faktury nebude nikdy žít sám, nikdy nezmění majitele, tedy jedná se o kompozici.

V UML spadají vztahy typu kompozice a sdílená agregace do obecnějšího pojmu agregace, tj. v UML se vztah agregace dělí na kompozici a sdílenou agregaci.

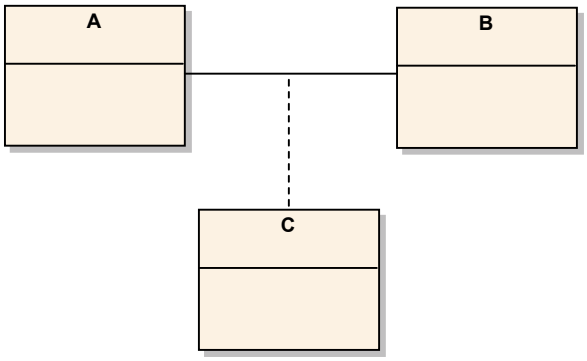
Pro modelování v AM důležitá skutečnost, že pokud se zavede v modelu AM vztah agregace mezi prvky (tj. buď sdílená agregace nebo kompozice), tak analytik dává designérovi pokyn, že instance celku (označená kosočtvercem) v programu ovládá své vnitřní členy, tj. používá je přímo svým chováním. Důsledkem zavedení agregace je proto viditelnost prvků od majitele ke svým částem a jejich přímé ovládání. Tato viditelnost se mapuje až do napsaného kódu, což je prakticky hmatatelný důsledek. Zavedení agregace v AM tak vede k přímému důsledku, kterým je to, že třída majitele musí bezpodmínečně mít ve viditelnosti třídu svých částí. Například vztah agregace vede při jednom určitém zvoleném mapování do databáze ke kaskádovitým operacím.

4.30 Asociativní třída

Při modelování v AM je mnohdy zapotřebí zavést takový typ informace, který má jednak charakter typu informace, tj. odpovídá třídě v AM podle předešlých kapitol, a přitom výskyty z této informace zprostředkovávají vztah mezi jinými výskyty informací. Takovýto typ informace se nazývá asociativní třída (ASSOCIATION CLASS). Z hlediska efektivního analytického modelování se doporučuje zavést takovýto typ informace, tj. asociativní třídu, vždy, když se vyskytne alespoň jeden z těchto případů:

- Nalezne se vztah mezi výskyty M:N. Tento vztah je chápán analyticky jako „nehotový“ a měl by se následně doplnit asociativní třídou.
- Nalezne se informace, která by měla být přiřazena ke každému výskytu vazby mezi informacemi, tj. existuje informace potřebující být „pověšena“ na výskyty vztahu mezi dvěma informacemi. Vzniká efekt, který nazveme jako vzor „efekt vánočních ozdob“.

Nejprve si uvedeme, jak se v UML asociativní třída znázorňuje a poté si vysvětlíme, jak je třeba ji chápat. Na následujícím obrázku je znázorněna asociativní (třída C) v syntaxi UML:



obrázek 63 Asociativní třída C

Vztah asociativní třídy si vysvětlíme na následujícím příkladu:

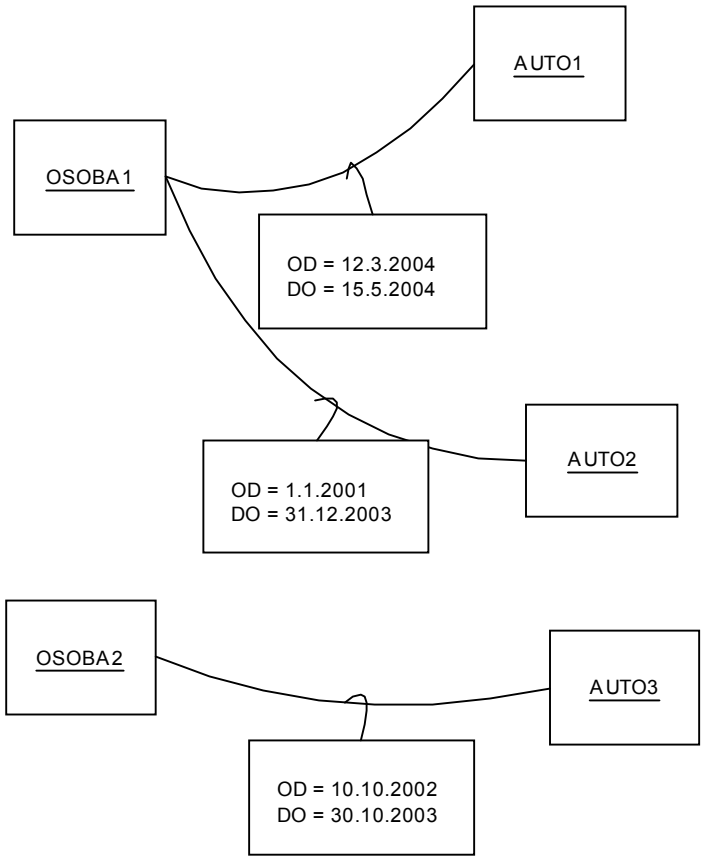
Máme za úkol vytvořit systém jednoduché evidence Osob (pro jednoduchost pouze fyzické osoby) a Aut. Charakteristickým atributem osoby je například rodné číslo, charakteristickým atributem auta je například SPZ, nebo VIN apod. Potřebujeme evidovat vztah mezi evidovanými auty a evidovanými osobami.

V jakém vztahu jsou instance osob a instance aut? Na jedné straně máme třídu Osoba, z ní evidované instance. Na straně druhé máme třídu Auto a z ní také evidované instance. Žádá se, aby pokud v systému „držíme“ instanci ze třídy Osoba, tak k ní dostaneme, tj. v programu nějak přejdeme k odpovídajícím instancím aut přiřazených k této instance osoby, a také obráceně v druhém směru.

Další otázka zní: Jaká je multiplicita v jednom a druhém směru takového přechodu od instancí k instancím? Pokud držíme „v ruce“ (samozřejmě myšleno v „programátorské ruce“ ☺) instanci osoby, tak k této osobě bude v evidenci přiřazeno evidentně N instancí aut. Obráceně sice v daném okamžiku je přiřazena pouze jedna osoba jako majitel pro dané auto, ale zde je v tom příkladu malý chyták: Dodáme slůvko „v čase“ a nikoliv evidence pouze pro aktuální stav. Samozřejmě evidováno v čase i s historií, tak k jednomu autu je přiřazeno N osob. Navíc však vznikla díky dodatku „v čase“ také nová informace „od - do“ jako dva datumy. Všimněme si, že tato informace „od do“ nepatří ani do auta (to by snad mohlo znamenat datum výroby a datum šrotu, což nás nezajímá), a ani do osoby (to by mohlo znamenat datum narození a datum úmrtí, což nás také nezajímá).

V tomto příkladu jsme dostali oba dva signály pro zavedení asociativní třídy: Existuje evidentně vztah mezi osobami a auty M:N (pokud se eviduje historie „v čase“), navíc

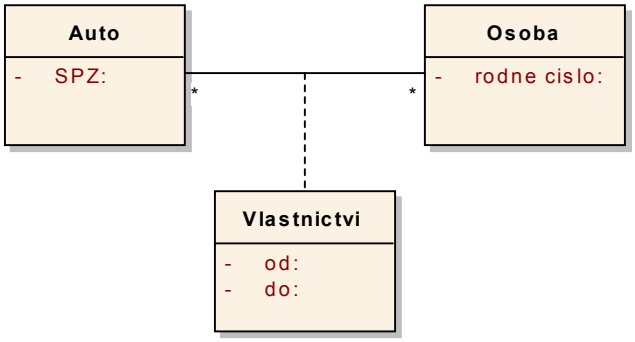
existuje také další informace nesoucí datumy „od do“, která vede k efektu „vánoční ozdoby“. Tento efekt se projeví tak, že na každou „evidenční spojnici“, která propojuje jedno nějaké příslušné auto s nějakou příslušnou osobou, potřebujeme doslova „pověsit“ jednu instanci nesoucí informaci „od do“ nějak takto:



obrázek 64 Efekt „vánočních ozdob“ v příkladu osob a aut

Uvedený obrázek je obdobou mírně upraveného instančního modelu a je třeba jej chápat jako „vymyšlený“ příklad možné evidence. Z obrázku je v této evidenci přímo čitelné, která osoba vlastnila které auto a od kdy do kdy jej vlastnila.

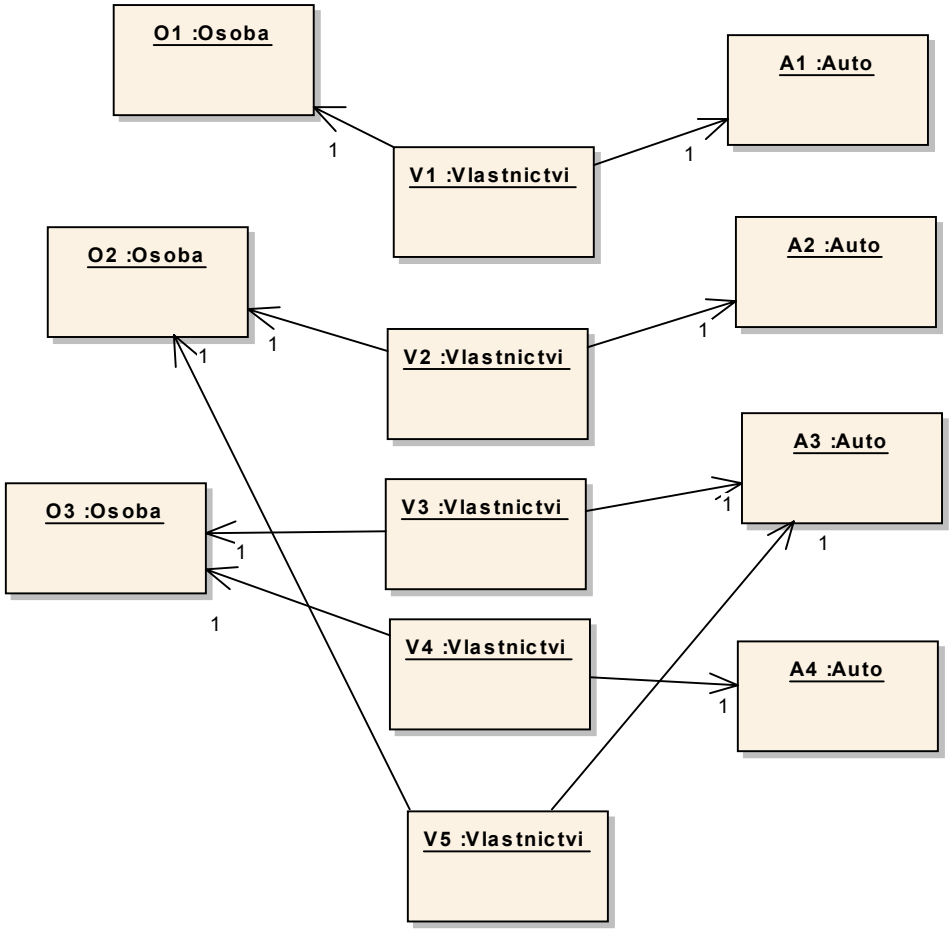
Pro instance „zavěšené“ na spojnicích musíme zavést třídu, nazvěme ji třeba Vlastnictví. Odpovídající model tříd bude potom vypadat takto:



obrázek 65 Model tříd pro evidenci osob a aut s asociativní třídou

Třída Vlastnictví je v tomto případě tzv. asociativní třída. Model máme sice namalovaný podle syntaxe UML, je však třeba jej vysvětlit konkrétně. Ukážeme si, jak asociativní třída funguje v praxi a jak je třeba ji chápat. Jak bylo řečeno, pro vysvětlení se nejlépe hodí instanční model, tedy použijme jej.

Pro asociativní třídu je charakteristické to, že její instance mají úplně stejnou strukturu, jako by z ní vycházely dvě číselníkové vazby do odpovídajících tříd. Představme si tedy, jako by ze třídy Vlastnictví vedly dvě šipky do třídy Auto a do třídy Osoba, samozřejmě pouze „jakoby“, protože takovou budou mít strukturu instance ze třídy Vlastnictví. Instance budou mít tuto strukturu:

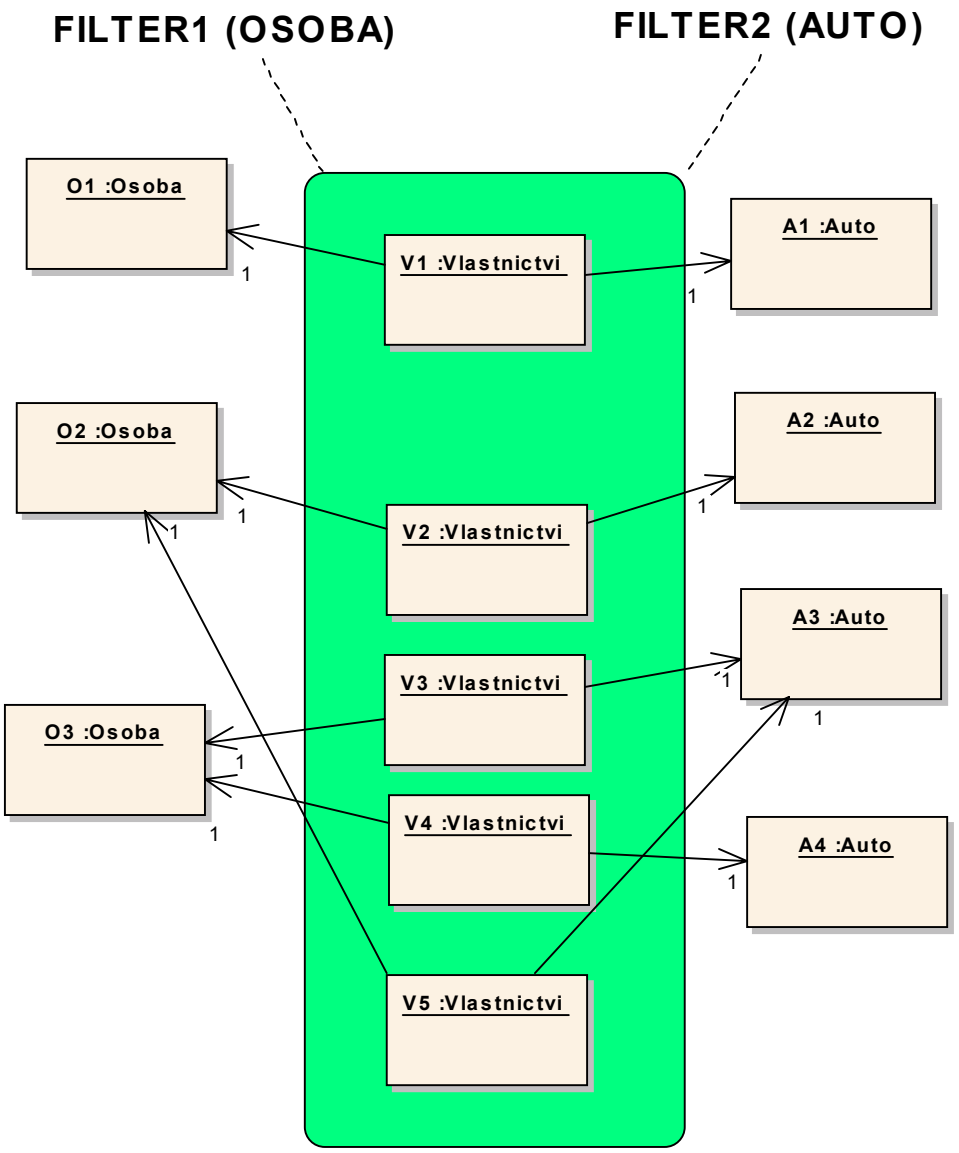


obrázek 66 Instanční model pro asociativní třídu je stejný jako dvojice běžných asociací

Naskýtá se pochopitelná otázka: Pokud jsou instance asociativní třídy strukturou stejné, jako kdybychom v modelu zavedl dvě běžné asociace číselníkové vazby (jednou od vlastnictví do osoby a podruhé do auta), jaký je tedy smysl zavést asociativní třídu namísto dvou běžných asociací? Jaký je v tom rozdíl?

Rozdíl spočívá v následujícím: Základní vlastností instancí asociativní třídy je to, že jejich seznam umožňuje přecházet od instancí jedné třídy k druhým a případně zpět. Děje se to takovým způsobem, že seznam instancí asociativní třídy musí být povinně vybaven funkcionalitou, která se chová jako „filtr“. Vstupním parametrem tohoto filtru je instance (resp. identifikátor této instance) ze třídy, na kterou si instance ukazuje. Filtr jako výsledek „vyplivne“ pouze ty ukazatele instancí asociativní třídy, které si na

tuto instanci ukazují. V předešlém obrázku bychom si mohli tuto funkcionalitu filtru představit takto:



obrázek 67 Filtry realizující propojení instancí díky instancím asociativní třídy

Představme si na předešlém obrázku, že seznam prvků Vlastnictví (zelená oblast) může pracovat s filtrem, nazvěme jej FILTER1, který má jako vstupní parametr instanci osoby (resp. její identifikátor). Tento filtr pracuje tak, že jako výsledek tohoto dotazu poskytne ukazatele na ty instance Vlastnictví, které si ukazují na tento vstupní parametr. Pokud je tedy například hodnota vstupního parametru filtru FILTER1

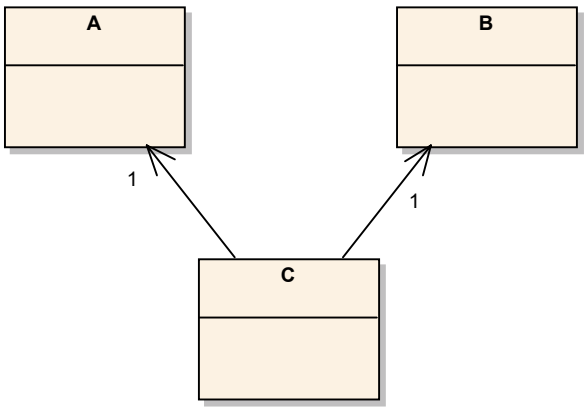
instance O1, tak výstupem filtru je ukazatel na instanci V1, analogicky pokud je vstupním parametrem instance O2, tak výstupem téhož filtru FILTER1 jsou instance V2 a V5, a tak dále. Protože instance Vlastnictví obsahují jednak informace „od do“ a navíc si ukazují na příslušné instance aut, můžeme požádat tyto instance Vlastnictví o vydání informací vlastnictví těchto aut od do. Díky tomuto filtru, který mimochodem není nic jiného, než již zmíněná kvalifikace vazby, se od instancí osob dostaneme k instancím aut přes instance vlastnictví. Úplně symetricky lze díky opačnému filtru FILTER2 analogicky přecházet od instancí aut k instancím osob, princip je úplně stejný, ale obrácený. Pomocí filtru FILTER2 dostaneme pro dané auto jako vstupní parametr ty instance Vlastnictví, které si ukazují na dané auto. Díky tomu, že každá instance Vlastnictví si ukazuje na osobu, dostaneme tak „toto auto bylo vlastněno od do těmito osobami“.

Tyto úvahy nyní dáme do obecnější podoby: Asociativní třída zavádí vztah mezi výskyty tříd A, B, C. Instance ze třídy C zprostředkovávají vztah mezi instance ze tříd A a B a tak, že:

- Každý výskyt z C používá jeden výskyt z A a jeden výskyt z B stejně jako v číselníkové vazbě.
- Třída C je multiinstanční. Seznam výskytů z C musí mít tu vlastnost, že lze od tohoto seznamu zadáním instance A získat ze seznamu pouze ty instance z C, které tuto instanci používají a naopak symetricky, zadáním instance B tomuto seznamu z výskytů C lze získat pouze ty instance C, které tuto instanci B používají. Jinak řečeno je zavedena povinná kvalifikace vazby z obou stran.

Těmito dvěma vlastnostmi umožňují výskyty z C zprostředkovat vztah mezi A a B a to tak, že klient pro obsluhu tohoto vztahu mezi A a B používá seznam z výskytů C. Při zadání vstupní podmínky (viz kvalifikace vazby) do tohoto seznamu se vstupním parametrem rovným „instance A“ mu tento seznam poskytne ta C, která tuto instanci používají. Protože instance C používá instanci B, lze získat jak informace v C, tak informace z příslušných instancí B viděných z C.

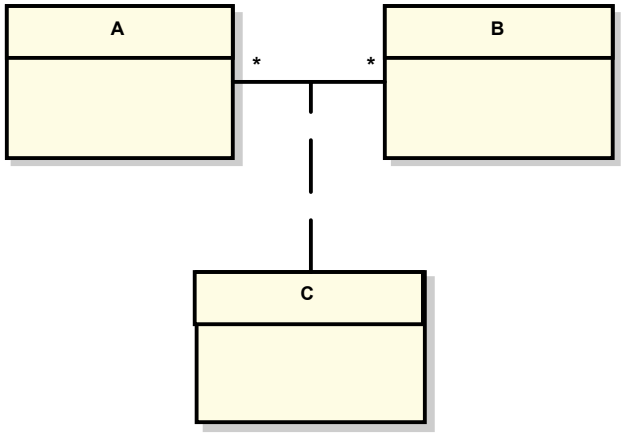
Pokud by se zavedla v modelu tříd namísto asociativní třídy taková třída, která má pouze dvě běžné asociace, tj. nějak takto:



obrázek 68 Dvě běžné asociace versus asociativní třída

tak z hlediska struktury se jedná o ekvivalentně stejný vztah jako u asociativní třídy. Rozdíl je však v chování instancí: V modelu na předešlém obrázku není řečeno, že existuje nějaký vztah mezi instancemi ze třídy A a instancemi ze třídy B, tj. že lze přes instance ze třídy C k těmto instancím přecházet. Předešlý model se dvěma běžnými asociacemi ukazuje pouze vlastnosti instance ze třídy C a nic víc, tečka. Pokud je vztah mezi A a B nalezen, potom je syntakticky přesnější převést vztah dvou běžných asociací na asociativní třídu, struktura instance typu C se přitom nezmění. Mnohdy je signálem pro nalezení asociativní třídy právě předešlý obrázek se dvěma běžnými asociacemi (signál „oslí uši“).

Je třeba podotknout, že mnohdy jsem byl v projektech svědkem, že autor modelu namaloval vztah jako předešlý obrázek se dvěmi číselníkovými vazbami, ale měl na mysli asociativní třídu. Protože v dalším analytickém popisu dynamického chování instancí dostatečně dobře popsal přechody mezi instancemi, systém byl následně v designu navržen správně a dobře naprogramován. Tento praktický poznatek mne dovedl k závěru, že tato záměna (dvě běžné asociace versus asociativní třída) není z hlediska návrhu fatální chybou a jedná se spíše o nepřesnost z hlediska notace UML. Samozřejmě, pokud chceme být přesní, měli bychom rozlišovat mezi oběma vztahy. Oproti předešlému obrázku pokud zavedeme asociativní třídu, tak model říká něco navíc – od instancí A k instancím B lze přecházet přes instance C:

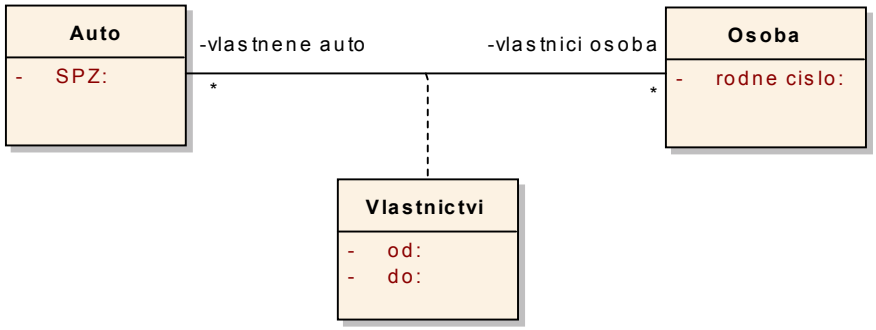


obrázek 69 Instance C má nejen definovanou strukturu, ale také propojuje instance ze tříd A a B

Je třeba upozornit na ještě jednu důležitou vlastnost asociativní třídy: Paradoxně díky asociativní třídě (na předešlém obrázku viz třída C) jsou třídy na jejích koncích (na předešlém obrázku viz třídy A a B) na sobě nezávislé a každá z nich „stojí“ samostatně, aniž by potřebovala druhou třídu. Jedině asociativní třída, která je propojuje, potřebuje obě dvě třídy.

Pro asociativní třídu (např. třída C na předešlém obrázku) je současně s touto vlastností příznačná i vlastnost druhá: K jednomu provázání mezi dvěma instancemi ze tříd A a B stačí založit jednu provazující instanci z asociativní třídy C. Pro pochopení této důležité vlastnosti se podívejme na obrázek 67. Pokud založíme novou instanci ze třídy Vlastnictví, tak jsme tímto jedním krokem provázali ihned dvě instance ze tříd Auto a Osoba. Nemusíme uvažovat o nějakých dvou krocích v jedné transakci v tom smyslu, že by se snad k nějakému autu přiřazovala instance z osob a obráceně v téže transakci by se přivázala k instanci osoby instance auta. Stačí jedna instance z C.

U asociativní třídy se na koncích také zavádějí role a značí, jakou roli hrají instance při přechodu od jedněch instancí k druhým přes instance asociativní třídy.
V předešlém příkladu jsou role evidentní: Jedna role by se mohla zavést jako „vlastnící osoba“ a druhá jako „vlastněné auto“ (pozor, role třídy ve vztahu odpovídá instancím, nikoliv třídám), například takto:



obrázek 70 Doplnění rolí u konců asociativní třídy

4.31 Mapování asociativní třídy do relační databáze

Mapování do relační databáze odpovídá mapování dvou běžných asociací, které jsme již brali. Do tabulky pocházející z asociativní třídy putují dva cizí klíče od každé tabulky jeden proti směru běžných asociací (šipek). V teorii databází se taková tabulka také nazývá analogicky asociativní tabulka. Předešlému modelu s analytickými třídami A, B a asociativní třídou C by následně odpovídaly tři tabulky s těmito klíči (P.K. značí primární klíč, F.K. značí cizí klíč):

TA :

ID_A P.K;

TB :

ID_B P.K. ;

TC :

ID_C P.K.,

ID_A F.K. z tabulky TA,

ID_B F.K. z tabulky TB;

Kromě toho musí designér navrhnout odpovídají povinné kvalifikace vazeb, které umožňují přechody mezi instancemi z jednoho konce asociativní třídy k instancím na druhém konci asociativní třídy. Tento požadavek využívá právě uvedených cizích klíčů v předešlém výčtu struktur tabulek a využívá SQL příkazů s klauzulí WHERE shody těchto klíčů.

4.32 Příklad na CONSTRAINT a abstraktní úrovně informačního systému

Vraťme se k příkladu s evidenci osob, aut a vlastnictví. Na tomto příkladu si lze kromě jiného velmi pěkně ukázat, jak uvažuje analytik a jak uvažuje designér.

Jako cvičení si jednu a tutéž otázku vyslovíme na dvou úrovních abstrakce, jednou na úrovni analytického modelování a podruhé na úrovni designu. Samozřejmě zkusme na tuto otázku také odpovědět.

Nejprve analytická formulace otázky: „Je možné, aby se v evidenci vyskytly dvě instance ze třídy Vlastnictví takové, že budou mít na koncích jak instanci ze třídy osoby tak instanci ze třídy auta obě stejné?“ Pro pochopení jak smyslu otázky, tak pro nalezení správné odpovědi si můžeme pomoci buď pohledem na obrázek 64 anebo na obrázek 67. Jako příklad takové situace, o které hovoří otázka, si na obrázek 64 představme, že by například mezi instancemi OSOBA1 a AUTO1 existovaly dvě spojnice a na nich dvě kartičky, na každé spojnici jedna.

Odpověď zní „takové dvě instance mohou existovat“ a v evidenci by se podchytila situace, kdy si dotyčná osoba koupí auto, poté jej prodá a následně znovu koupí, ať už z důvodu spekulace anebo proto, že se mu stýskalo po jeho „miláčkovi“.

Tutéž otázku si položme jako designéři relační databáze: Bude dvojice cizích klíčů ID_AUTO a ID_OSOBA, která se vyskytuje jako dvojice cizích klíčů, v tabulce TVLASTNICTVI, unikátní anebo ne? Odpověď zní opět nikoliv a důvod je úplně stejný, jako v předešlém odstavci.

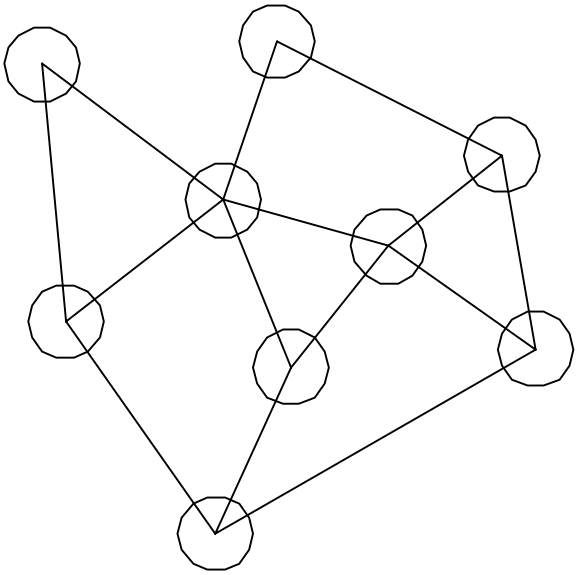
I když se to zdá v této chvíli pro „rozené programátory“ neuvěřitelné, první formulace otázky je jednodušší a srozumitelnější. Pokud se vám jeví, že opak je pravdou a formulace s klíči je „lepší“, tak jste „hodně ponoření do relační technologie“. Když se totiž uvažuje o unikátnosti cizích klíčů, automaticky v mysli designéra proběhne tato úvaha rychlostí blesku: Shoda klíčů by znamenala dva stejné záznamy jak v tabulce

auta, tak v tabulce osoby, tj. totéž auto a tatáž osoba, a to jde, protože proč by nemohlo jedno auto patřit téže osobě dvakrát (samozřejmě s historií v čase). Jenomže poslední část úvahy již spadá do abstraktní úrovně analytického modelování!

Navíc zkusme si představit „normálního“ člověka, externího konzultanta, nebo prostě někoho, kdo se vůbec nezajímá o relační databázi. Chceme formulovat předešlou otázku, protože na ni (jakože) neznáme odpověď, ale zná ji konzultant. Naučíme snad dotyčného základům teorie databází a co jsou to cizí klíče? Ne, musíme použít analytickou formulaci s instancemi, která je navíc obecná a nezávislá na technologii. Analytická formulace totiž vyjadřuje pouze myšlenku o evidenci a nic víc.

4.33 Příklad na asociativní třídu v teorii grafu

V jedné firmě řešili zajímavý evidenční systém, nazývaný vzletně „Optimalizace svozu odpadu“. V podstatě se jednalo o jízdu popelářských vozů městem. Představme si, že na mapě města existují křižovatky, mezi nimi existují úseky a na nich adresy. Někde jsou umístěny garáže, z nich vyjedou vozy, projedou adresy a svezou odpad na skládky. Obrázek plánu města, což vlastně odpovídá zobrazenému instančnímu modelu na obrazovce, by mohl vypadat nějak takto:

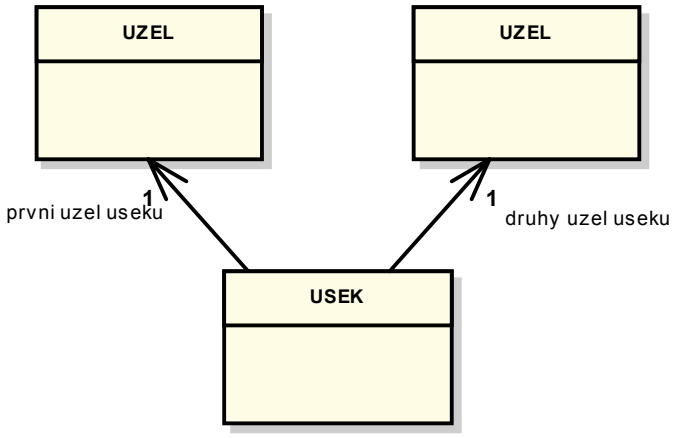


obrázek 71 Instance křižovatek

Jak vidět, matematicky se jedná o teorii grafu.

Smyslem tohoto evidenčního systému je snížit náklady průjezdu vozů městem. Když mi poprvé o této úloze řekli, napadla mne ihned otázka: „A má to vůbec řešení? Vždyť to silně připomíná úlohu obchodního cestujícího, v tomto případě dokonce N obchodních cestujících!“ Odpověď zněla: „Máme svoje know-how, které je schopno dostatečně náklady snižovat, nehledáme ideální řešení...“

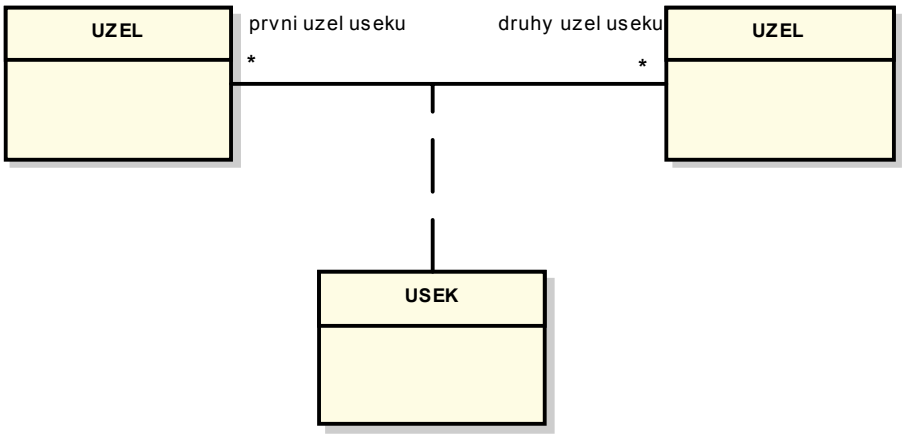
Pro uvedenou úlohu je důležité zavést evidenci vrcholů grafu, v dané aplikaci byla za tím účelem zavedena třída Uzel. Kromě toho existují hrany grafu, které spojují dené vrcholy, za tím účelem byla zavedena druhá třída s názvem Úsek. Pokud se podíváme na strukturu instancí, zjistíme, že každý úsek má dva vrcholy, které nemá v kompozici (oba konce se při zadání nového úseku musí vybrat a dosadit). V prvním přiblížení bychom mohli navrhnout tento model:



obrázek 72 První návrh modelu tříd pro evidenci grafu

Každý úsek má dva úseky a ty nejsou jeho kompozicí, ale dosazují se do něj výběrem mezi ostatními uzly. Jeden uzel je chápán jako první uzel úseku (na jeho jednom konci) a druhý je chápán jako druhý uzel úseku (na jeho druhém konci), od toho vyplývají i názvy rolí na koncích „číselníkové vazby“.

Několik dnů jsme v dané firmě při školení používali tento model, až se zvedla ruka jednoho posluchače a byl vznesen tento dotaz: „Nemělo by se to spíše malovat takto?“:

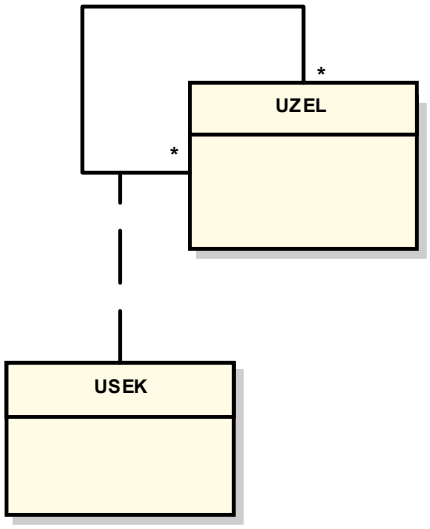


obrázek 73 Jiný model grafu

I když oba modely jsou si velmi podobné (v obou modelech instance ze třídy Úsek „vidí“ dvě své instance Uzlu na svých koncích), tak tento druhý model se jeví jako přesnější. Vyjadřuje totiž tu skutečnost, že sice instance ze třídy Usek „vidí“ dvě své instance Uzlu na svých koncích, ale navíc přes instance úseků se lze dostávat od

jedněch instancí uzlů k druhým instancím přes společné úseky, což reprezentuje přechod k sousedním uzlům v grafu. Jenom je třeba upozornit, že pokud chceme opravdu všechny sousední uzly daného uzlu, musíme provést filtraci z obou stran a poté tyto dvě množiny sloučit, protože nikdy nevíme, v jakém pořadí do kterých konců je uzel umístěn.

Musíme zdůraznit ještě tu skutečnost, že uvedený model lze namalovat úplně ekvivalentně i takto:



obrázek 74 Ekvivalentně zapsaný model jako předešlý obrázek

Jediný rozdíl mezi oběma předešlými obrázky je ten, že na druhém obrázku jsme třídu Uzel namalovali pouze jednou. Oba obrázky však vyjadřují tentýž model.

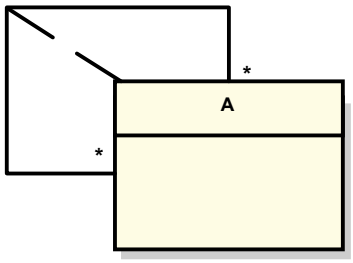
4.34 Příklad jako hádanka „Co to je?“

Určitě znáte hádanky typu: „Co to je?“, například: „Má to dvě ruce, šest nohou a tři oči?“

Odpověď: „Jan Žižka na koni.“

Dejme si podobnou hádanku:

S kolegy jsme řešili nějakou evidenci, a vyšel nám tento model:



obrázek 75 Zajímavý model s asociativní třídou „sama na sebe“

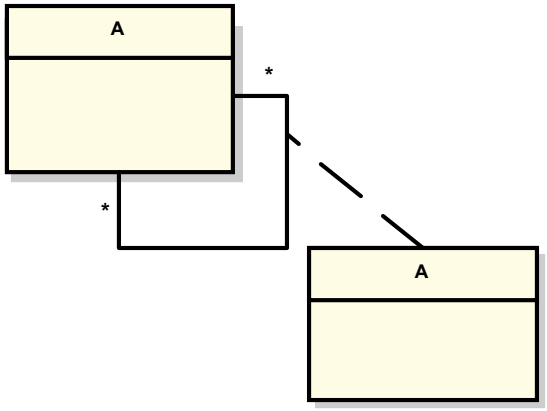
V podstatě se jedná o asociativní třídu sama na sebe.

Hádanka zní: Co jsme řešili za evidenci? Při řešení této úlohy na cvičeních při školení in-house hádali pracovníci „všechno možné“ a většinou po několika návrzích sami dospěli ke správné odpovědi. Než budete číst dále, zkuste se zamyslet a „uhádnout“ o co jde.

Odpověď zní (už jste se sami zamysleli…?):

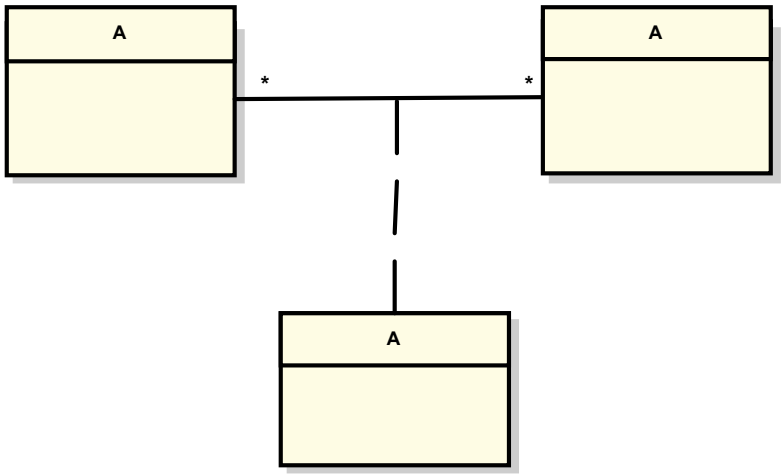
„… řešili jsme rodokmen…“

Pro vysvětlení si je třeba uvědomit, že uvedený zápis je „nejvíce“ hutný, jaký může být. Existuje další plně ekvivalentní zápis téže stejné myšlenky a téhož modelu:



obrázek 76 Ekvivalentní zápis předešlého modelu

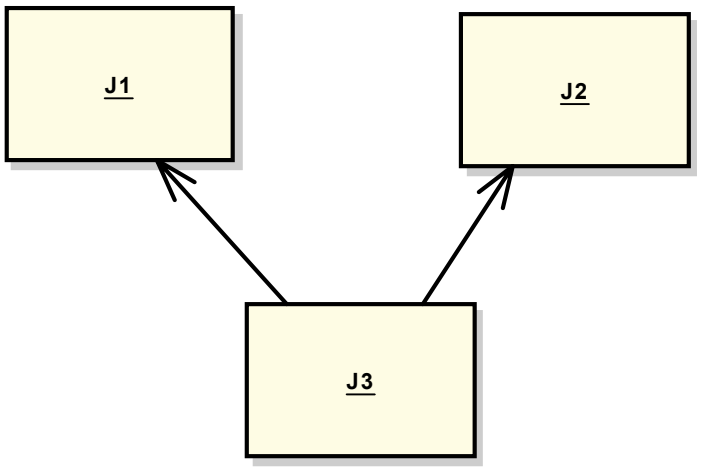
Ekvivalence zápisů ještě nekončí. Stejný model se dá zapsat ještě dalším ekvivalentním způsobem takto:



obrázek 77 Další ekvivalentní zápis téhož modelu

Tento poslední zápis je, zdá se, nejpřehlednější.

Pro vysvětlení, jak funguje zmíněná evidence rodokmenu, je nejlepší použít instanční model. Třídu v rodokmenu nazvěme třída Jedinec, protože jím může být např. šlechtic, kůň nebo pes. Jak bude vypadat instanční model evidovaných jedinců rodokmenu podle tohoto modelu? Začněme nejjednodušší evidencí tří jedinců takto:

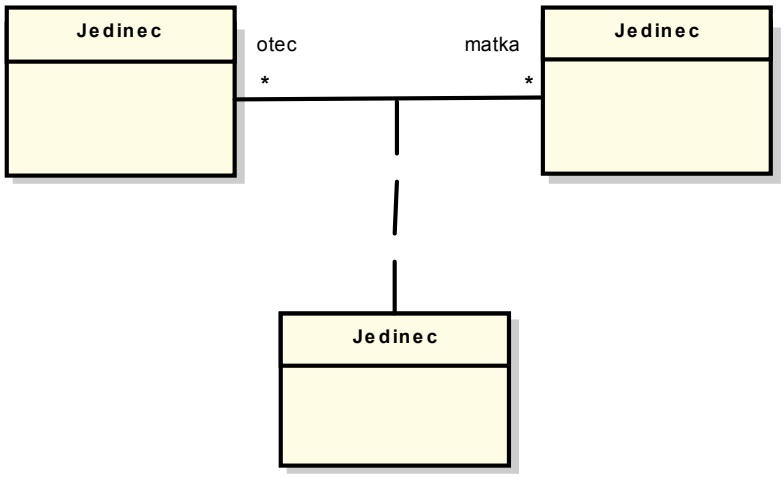


obrázek 78 Evidence tří jedinců v rodokmenu

Z hlediska syntaxe UML šipky na předešlém obrázku nereprezentují vztah mezi třídami, ale již konkrétní „evidenční spojnici“ mezi instancemi (v UML nazývaný také jako LINK) jako realizace tohoto vztahu. Tři evidovaní jedinci vytvářejí základní

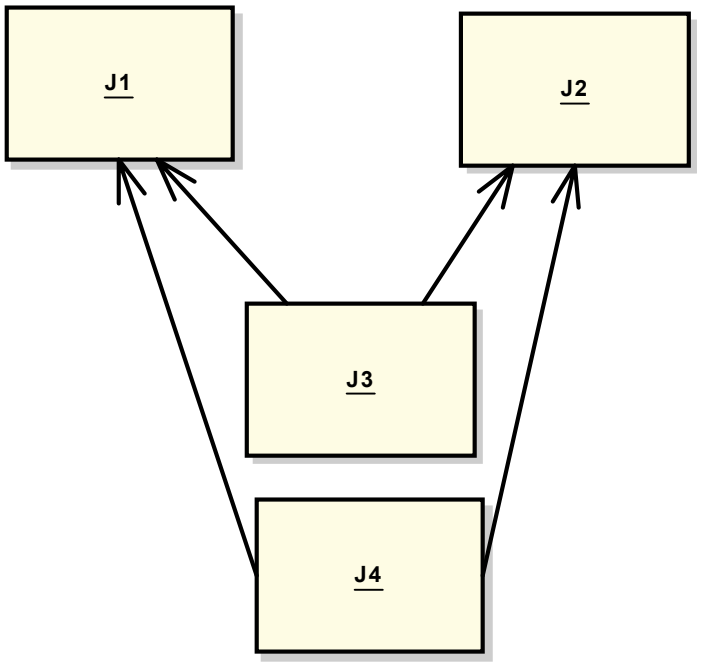
strukturu rodokmenu: Otec, matka a dítě. Následující věta zní trochu podivně, ale je tomu tak: V evidenci instance dítěte „vidí“ svou matku a „vidí“ svého otce. V jakém vztahu jsou instance na koncích asociativní třídy, tj. které jsou spojené v evidenci přes asociativní třídu? Instance na koncích asociativní třídy jsou ty, které „spolu něco měly“ a „mělo to následek“ (☺), hned musíme dodat, že se jedná pouze o biologický rodokmen s biologickými následky.

Při znalosti , o co jde v instančním modelu, je možné nyní přidat i role v modelu:



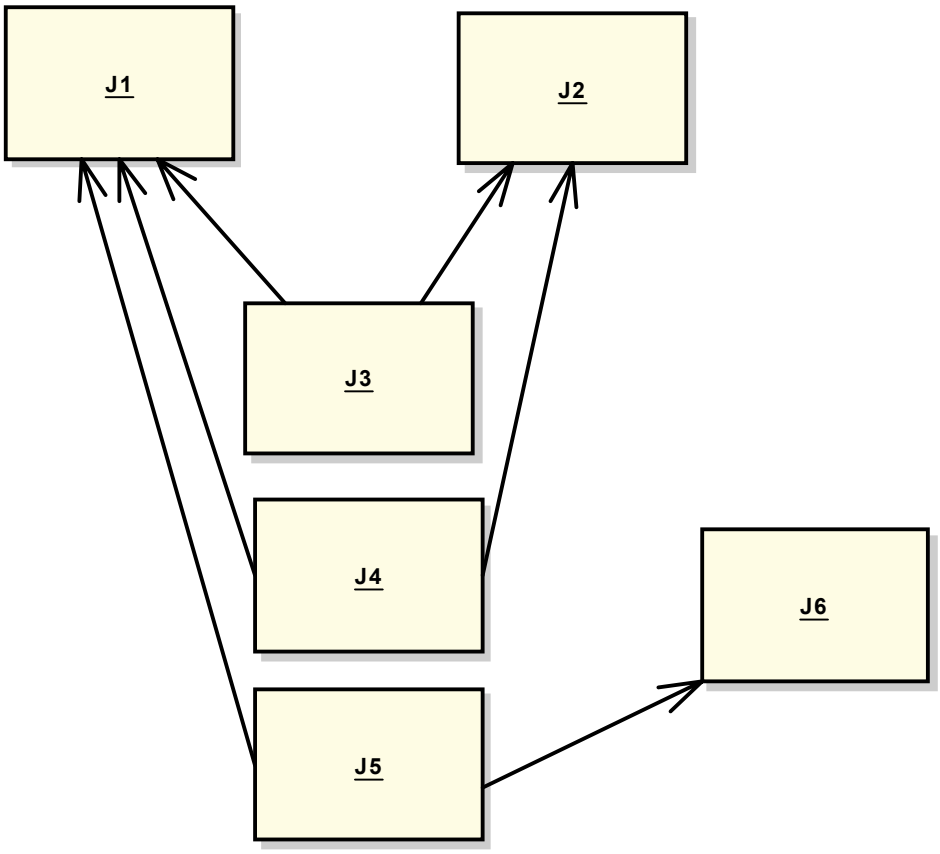
obrázek 79 Rodokmen i s rolemi

V modelu lze vystopovat i další vztahy, jako např. vztah přímých sourozenců:



obrázek 80 Vztah přímých sourozenců J3 a J4

anebo také i další vztahy, jako například nevlastní sourozenci:



obrázek 81 Vztah nevlastního sourozence J5 vůči J3 a J4

Z uvedených příkladů vyplývá, že takováto evidence biologického rodokmenu by mohla mít své konkrétní využití v genetice, kdy se budoucí rodiče mohou dotázat genetiků na rizika pro jejich potomky.

Uvedené modely zobrazují evidenci, která bude nakonec nějak realizována v programu, tj. designér bude model podle diagramu na obrázku viz obrázek 79 mapovat do svého prostředí.

Jako příklad mapování do designu si uveďme mapování do datových struktur relační databáze. Vznikne pouze jedna tabulka TJEDINEC s dvěma cizími klíči „sama na sebe“, nazvěme je ID_OTEC, ID_MATKA:

Tabulka TJEDINEC:

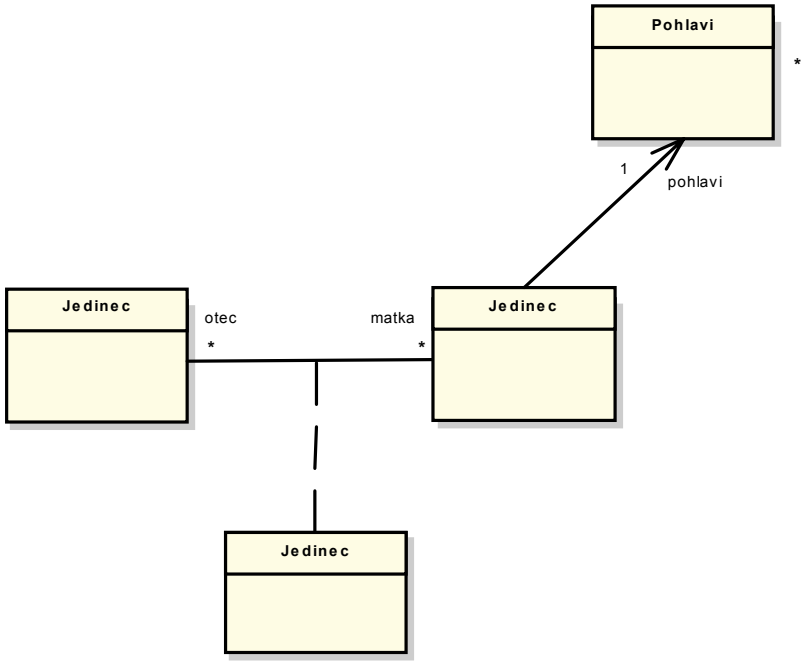
ID_JEDINEC P.K.

ID_OTEC F.K.

ID_MATKA F.K.

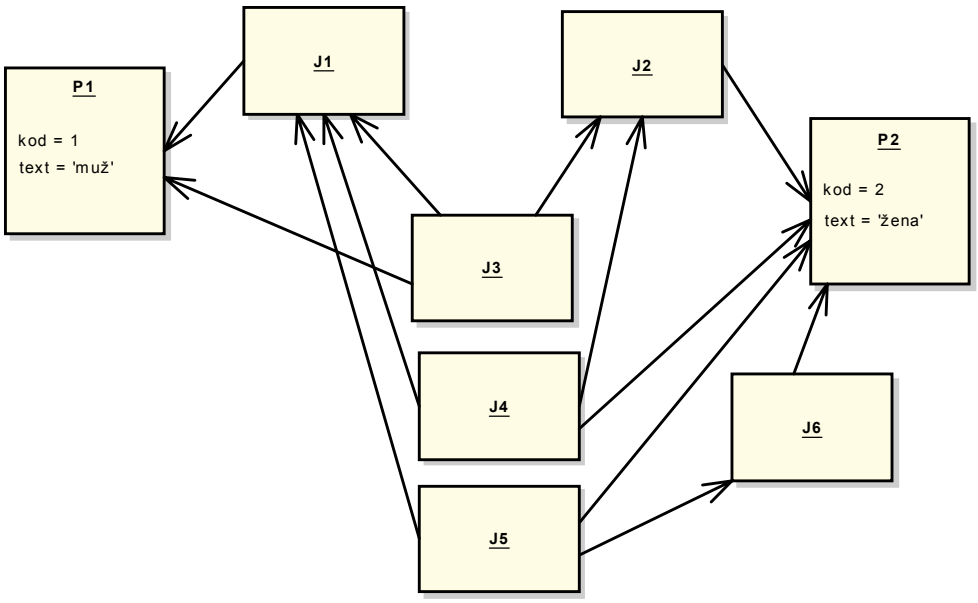
Zajímavá otázka, která může být formulována trochu podivně: Jak dostanu všechny moje děti? Tedy přesněji: Pokud je v evidenci „držena“ instance ze třídy Jedinec (například na předešlém obrázku instance J1), tak jak dostaneme všechny jeho přímé potomky v prvním koleně? Jak vidět, musíme provést kvalifikaci vazby jak přes vazbu „na otce“, tak přes vazbu „na matku“ (pohlaví není evidováno!). Jinak řečeno, musí se dostat všechny instance, které danou instanci „vidí“ buď jako otce anebo jako matku. Totéž vyjádřeno v designu relační databáze a je tedy konkretizováno do technologie RDB, zní takto: Pokud „držíme“ nějaký identifikátor jedince ID_JEDINEC, tak dostat všechny děti znamená provést nad tabulkou dva dotazy nad cizím klíči jak ID_OTEC, tak ID_MATKA rovnými této držené hodnotě.

Při těchto dvou dotazech „přes matku“ a „přes otce“ je jasné, že jeden z těchto dvou dotazů by měl vyjít určitě jako prázdný. To nás může vést k myšlence vylepšit evidenci tak, že budeme evidovat pohlaví. Zavedeme proto klasický číselník pohlaví, který bude mít dva prvky. Evidence by potom mohla vypadat takto:



obrázek 82 Rodokmen i s evidovaným pohlavím

Instanční model by potom mohl vypadat například takto:



obrázek 83 Instanční model rodokmenu i s evidencí pohlavím

Na předešlém obrázku jsou zavedeny i dvě instance číselníku Pohlaví. Všimněme si, že kvalifikací vazby do tohoto číselníku zjistíme, že instance jedinců s vlastností „muž“ jsou J1 a J3 a jedinci s vlastností „žena“ jsou všechny ostatní. Tuto informaci bychom v systému dostali opět tzv. kvalifikací vazby (např. dej mi všechny instance, které si ukazují na P1 resp. dej mi všechny instance, které si ukazují na P2).

Existují nyní dvě možnosti pro „získání dětí“ daného jedince a přitom využít evidence Pohlaví. U obou řešení se vyhneme jednomu prázdnému dotazu, který v našem příkladu nastal.

První z řešení se nabízí jako jednodušší a odpovídá „klasickému“ strukturálnímu pojetí: Uvnitř jedince se ve funkcionalitách zavede klasický přepínač typu „if … else“ nebo „switch“ resp. „select case“ apod. Tento přepínač se aplikuje na ukazatel na instanci pohlaví (viz předešlý obrázek - např. na ID nebo kód). Dotaz potom vypadá symbolicky nějak takto: „Pokud si ukazuji do číselníku pohlaví na instanci muž, tak se provede výběr jedna, pokud na instanci žena tak výběr dva“. Tento přepínač odpovídá strukturálnímu přístupu.

Existuje druhé řešení, které nyní dopodrobna nemůžeme probrat, protože nemáme k dispozici veškerou syntaxi CLASS MODELU v UML. Zatím se o tomto řešení zmíníme pouze okrajově. Toto řešení bude více jasné pro ty z vás, kteří mají zkušenosti s objektovým programováním. Již v analytickém modelování se zavedou dva podtypy typu Jedinec (dědicové), jeden bude typu Muž a druhý Žena. Tyto podtypy se liší polymorfním chováním „vyhledávání dětí“, každý podtyp má svůj

algoritmus, výstup kvalifikace jeden přes jednu roli, druhý přes druhou roli. Například v hybridním systému v jazyce OOP (JAVA, DOT NET DEPLHI apod.) by se toto řešení projevilo tak, že dva dědicové třídy CJEDINEC, což jsou třída CMUZ a třída CZENA, by přepisovaly metodu `GetChilds`, ve které by byl volán v datové vrstvě SQL příkaz buď přes jeden nebo přes druhý klíč. Přepínač je zaměněn algoritmem: „Typ by už sám věděl“. Navíc by bylo vhodné použít pro generování instancí některý z DESIGN PATTERNS, například PROTOTYPE, aby ani při volání konstruktoru nebyl použit strukturální přepínač.

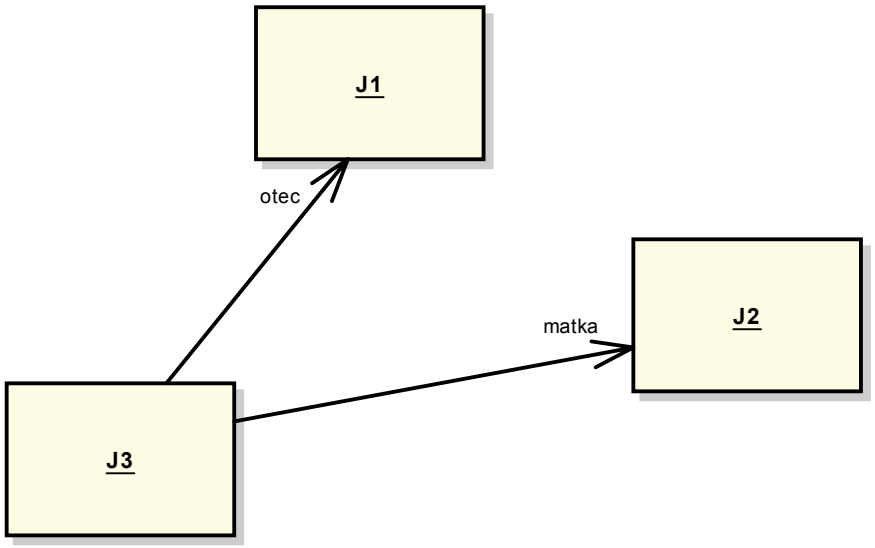
K tomuto příkladu se vrátíme, až probereme vztah GENERALISATION – SPECIALISATION, který k tomuto výkladu nutně potřebujeme.

4.35 Stejný příklad podruhé a jinak

Je možné, že řešení s „přepínačem mezi mužem a ženou“ (ať už datovým anebo typovým „více objektovým“) se nám prostě nelíbí a pokusíme se evidenci navrhnout jinak. Samozřejmě už to nebude příklad na „asociativní třídu sama na sebe“, ale příklad jiný, ale i tak velmi poučný.

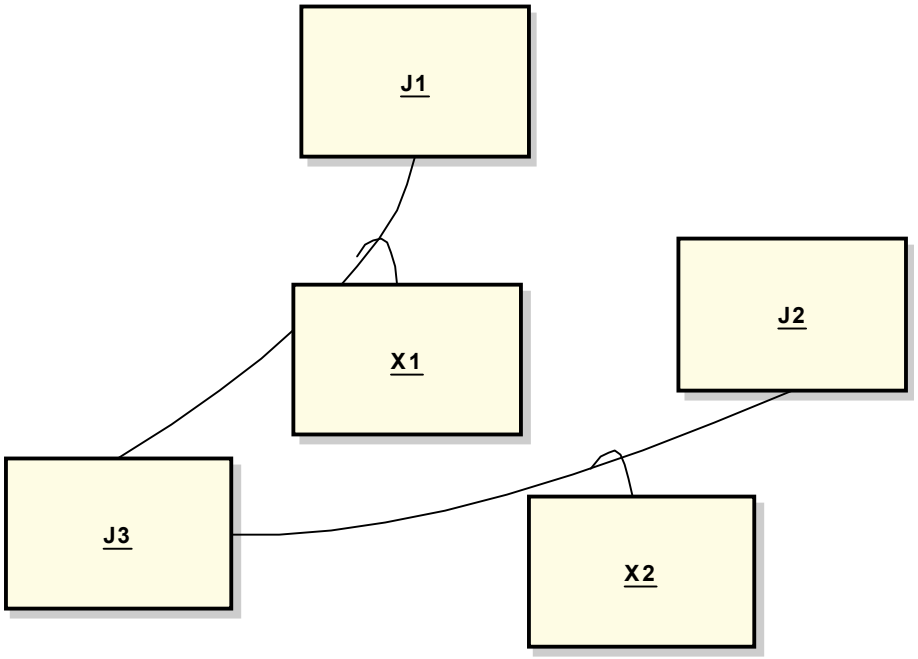
Cílem je tedy „zbavit se“ dvojího průchodu přes ukazatele, přepínat se přes roli matky a přes roli otce. Když se nad tímto požadavkem hlouběji zamyslíme, jedná se vlastně o „efekt vánočních ozdob“. Chceme, aby výběrová podmínka byla přes oba ukazatele „najednou“. Jinak řečeno chceme, aby pohled na oba ukazatele od dítěte, tj. pohled „otec“ a pohled „matka“ již nebyly od sebe odlišeny jako dvě vedle sebe stojící odlišené instance, ale byl to jeden pohled „nahoru“ jakoby v seznamu.

Tomu odpovídá namísto „dvou instancí vedle sebe“ učinit vazbu přes asociativní třídu, která je dá dohromady, tj. původní pohled „vidím otce, vidím matku“ dom jednoho pohledu nějak takto:



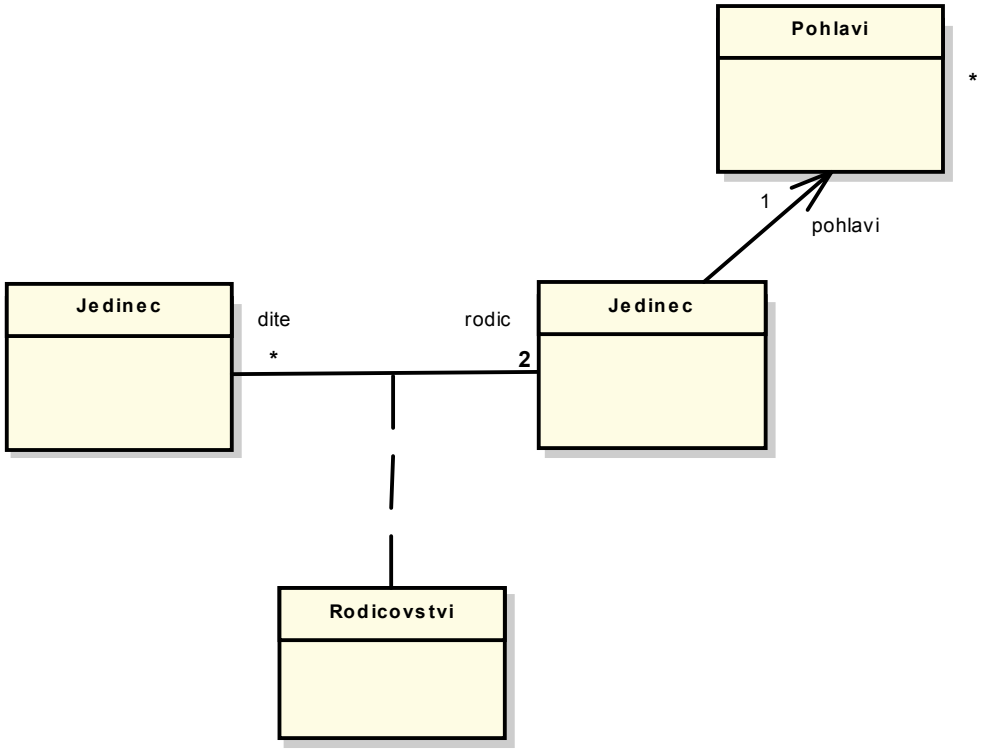
obrázek 84 Původní model s třídou Jedinec jako asociativní třídou

se změní na tento model „s vánočními ozdobami“:



obrázek 85 Jiný instanční model rodokmenu

Rozdíl mezi předešlými dvěma modely není „až tak velký“, v obou případech jsou provázány instance rodičů a dětí, ale děje se tak jinak: V modelu „vidím otce, vidím matku“ se jedná o přímou viditelnost těchto dvou instancí od dítěte k oběma instancím. Zde v předešlém obrázku existuje asociativní třída (zatím se projevila instancemi X1, X2), která tyto instance dítěte a rodičů provazuje. Instance J1 a J2 jsou opět rodiče instance J3 jako v předešlém modelu. Třídu instancí X1 a X2, která provazuje dítě a rodiče nazvěme Rodičovství. Model tříd potom vypadá takto:



obrázek 86 Jiný analytický návrh modelu tříd rodokmenu

V tomto druhém modelu lze nyní dostat „všechny moje děti“ opravdu přes jednu jedinou vazbu průchodem instancemi ze třídy Rodičovství. Všimněme si ještě zajímavé multiplicity na konci asociativní třídy u role „rodič“ rovnou dvěma.

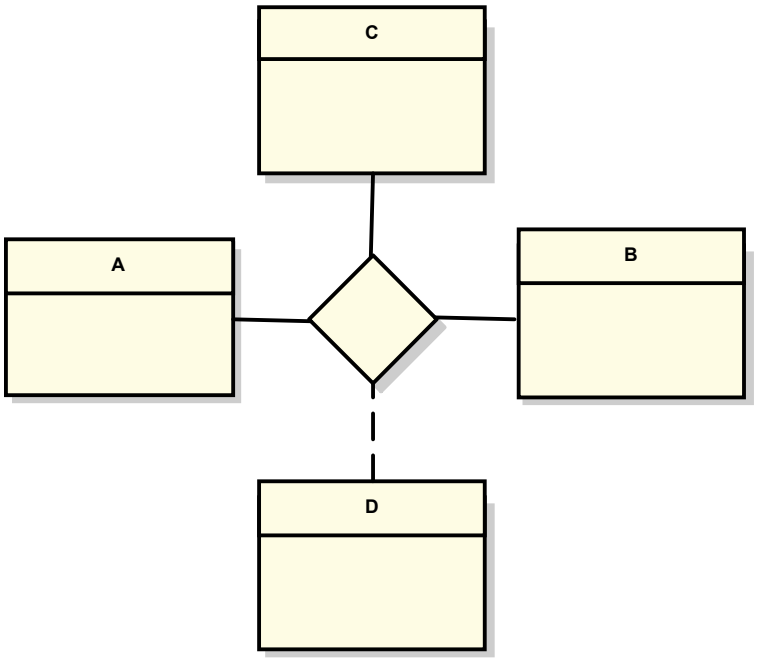
4.36 Běžná asociace ku N

V kapitole pojednávající o běžné asociaci (číselníkové vazbě) bylo uvedeno, že vztah běžné asociace se ve valné většině případů vyskytuje ve vztahu ku 1 (jeden ukazatel do číselníku). Pokud se nalezne vztah násobné asociace od jedné instance k N instancím (a nejedná se o agregaci), doporučuje se, aby se tento vztah automaticky převedl na vztah přes asociativní třídu s omezením na jedné straně ku 1. Návrh je tak otevřen k možnému a častému požadavku na rozšíření obecnějšího vztahu M ku N, což většinou bývá později požadováno.

4.37 Násobná asociativní třída

Násobnou asociativní třídu (N-ARY ASSOCIATION CLASS) lze chápat jako zobecnění asociativní třídy pro více než dva konce asociativní třídy. Zatímco asociativní třída C dala do vztahu dvě informace A a B, násobná asociativní třída dává do vztahu vícero informací než dvě (např. tři, čtyři atd.).

V syntaxi se používá kosočtverec:

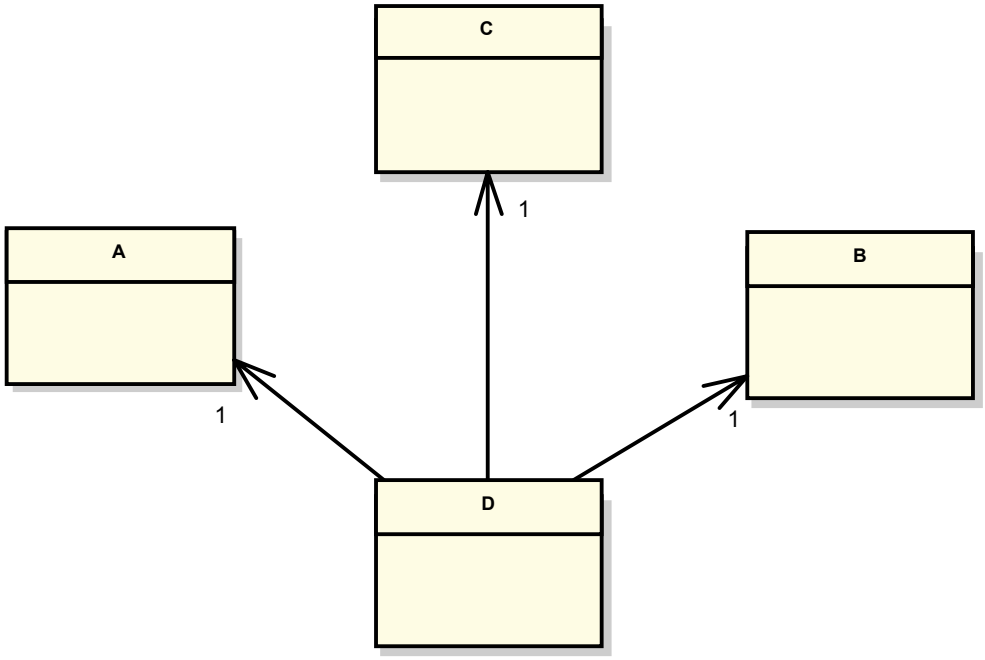


obrázek 87 Zavedení násobné asociativní třídy

Třída D na předešlém obrázku je násobnou asociativní třídou, avšak nespojuje dvě třídy, ale v tomto případě tři třídy A, B a C. Význam násobné asociativní třídy je úplně stejný jako u předešle zavedené asociativní třídy, pouze počet konců asociace je vyšší. Z hlediska matematické přesnosti syntaxe se tvůrci UML dopustili „malé nepřesnosti“ v tom smyslu, že násobná asociativní třída není nic nového, přesněji řečeno, asociativní třída probraná v předešlé kapitole je zvláštní případ násobné asociativní třídy pro 2 konce. Pokud bychom na předešlém obrázku vypustili např. třídu C, povaha vazby by zůstala zachována i pro A a B, ale museli bychom vypustit také kosočtverec (jednalo by se o asociativní třídu z předešlých kapitol).

U násobné asociativní třídy D mají její výskyty vlastnost, že každý z těchto výskytů používá jednu instanci z A, jednu instanci z B a jednu instanci z C podobně, jako by z této instance vedly tři šipky. Vztah je tedy opět velmi podobný jako tři běžné asociace z D do tří entit, ale navíc lze přecházet různým způsobem od jedněch instancí z jedné třídy k jiným instancím přes funkcionalitu filtrů seznamu z výskytů D. Výskyty D mohou nést také svou nějakou informaci. Naplnění instance na koncích násobné asociativní třídy nastává výběrem ze seznamů stejně jako u běžných asociací.

Protože strukturou je výskyt násobné asociativní třídy shodný se strukturou N běžných asociací, zavádí se někdy namísto násobné asociativní třídy s určitou mírou nepřesnosti N běžných asociací takto:



obrázek 88 Tři běžné asociace (číselníkové vazby) namísto násobné asociativní třídy, trcohu nepřesné, ale často používané

Pokud je popis scénářů v pořádku, nejde o příliš závažnou chybu, jedná se o sémantickou chybu UML.

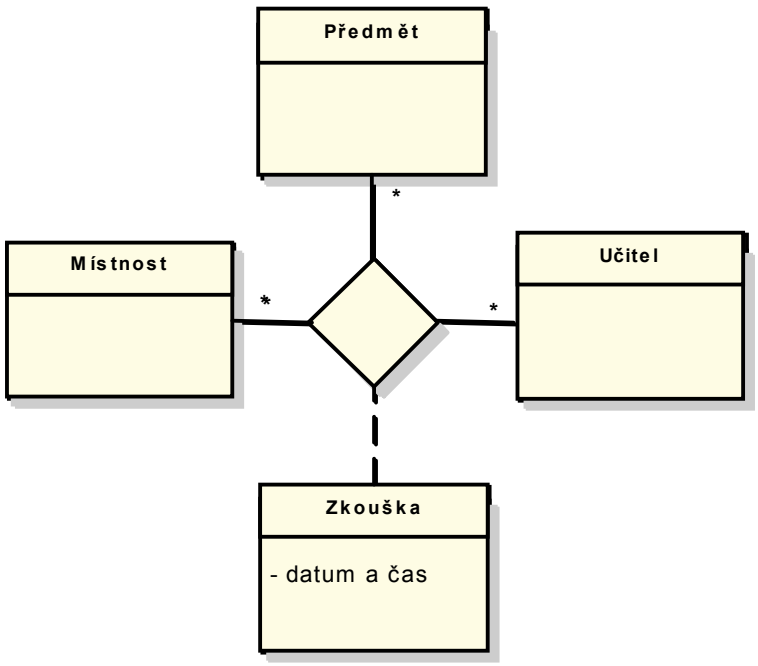
4.38 Příklad na násobnou asociativní třídu

Budeme řešit jednoduchou evidenci informačního systému vysoké školy pro intranet. Studenti se budou chtít přihlásit ke zkouškám. Tyto zkoušky budou zavedeny v systému, student se jako user přihlásí do systému, zkoušku si najde a přihlásí se na ni.

Bude se evidovat to, že daná zkouška se koná v určité místnosti, že se koná z určitého předmětu a že ji provede určený učitel. Zkouška se koná k určitému datu a v určité hodině.

Pro založení výskytu zkoušky je třeba vybrat existující evidovanou místnost ze seznamu evidovaných místností, podobně je třeba vybrat existující předmět ze

seznamu předmětů a je třeba vybrat existujícího učitele ze seznamu učitelů. Z toho důvodu to na první pohled vypadá, že zkouška si „ukazuje“ třemi číselníkovými vazbami do tří seznamů stejně, jako ukazuje obrázek v předešlé kapitole. Raději však zavedeme násobnou asociativní třídu např. takto:



obrázek 89 Zkouška jako násobná asociativní třída

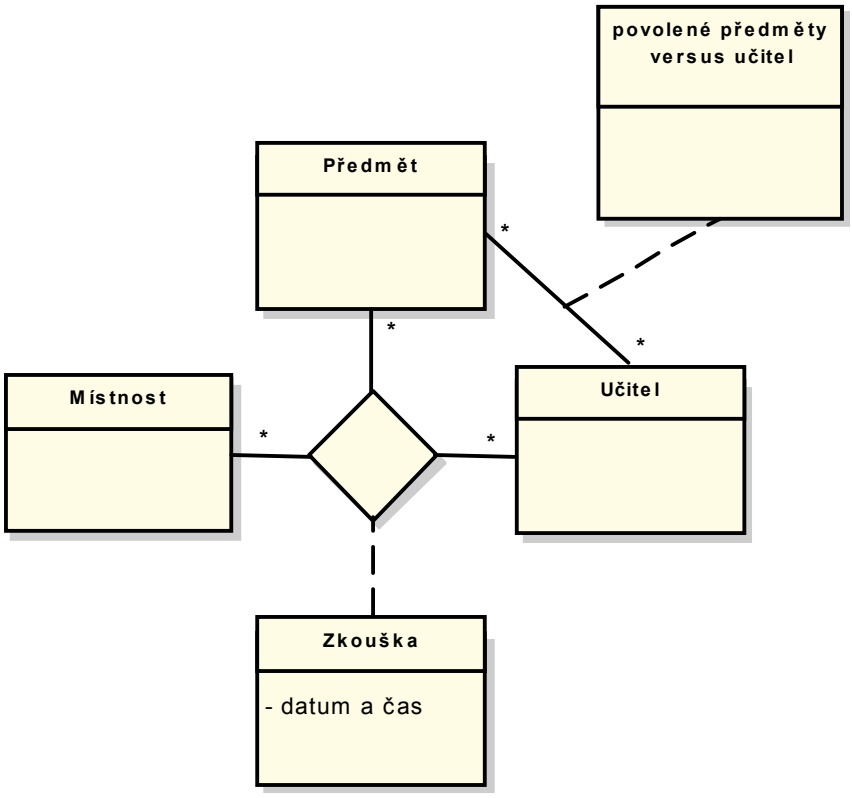
I když jsme zavedli násobnou asociativní třídu, i tak si „v duchu“ představujeme, že každá instance ze třídy zkouška si ukazuje na jednu vybranou instanci ze seznamu místností, na jednu instanci ze seznamu předmětů a na jednu instanci ze seznamů učitelů. Rozdíl oproti třem šipkám tedy není až tak velký. Předešlým obrázkem dáváme navíc najevo tu skutečnost, že se od jedněch instancí z jedné strany lze přes instance zkoušky dostávat k instancím na druhých koncích. Můžeme například díky této vazbě vyžadovat různé kombinace: Všechny zkoušky a místnosti, které provádí daný učitel, nebo všechny zkoušky v dané místnosti a kdo tam zkouší atd. Většinou analytik vyjádří vcelku obecný požadavek možnosti přechodů pomocí asociativní třídy, který následně konkretizuje ve scénářích chování instancí.

4.39 Příklad na zkoušku na vysoké škole rozvedený dále do dalších podrobností

Daný příklad můžeme ještě dále rozvádět jako pěkné cvičení. Ukážeme si na něm, jak by mohl vypadat rozhovor analytika s konzultantem a také poukážeme na jeden velmi důležitý požadavek analytického modelování, kterým je dodržení analytické „čistoty pojmů“.

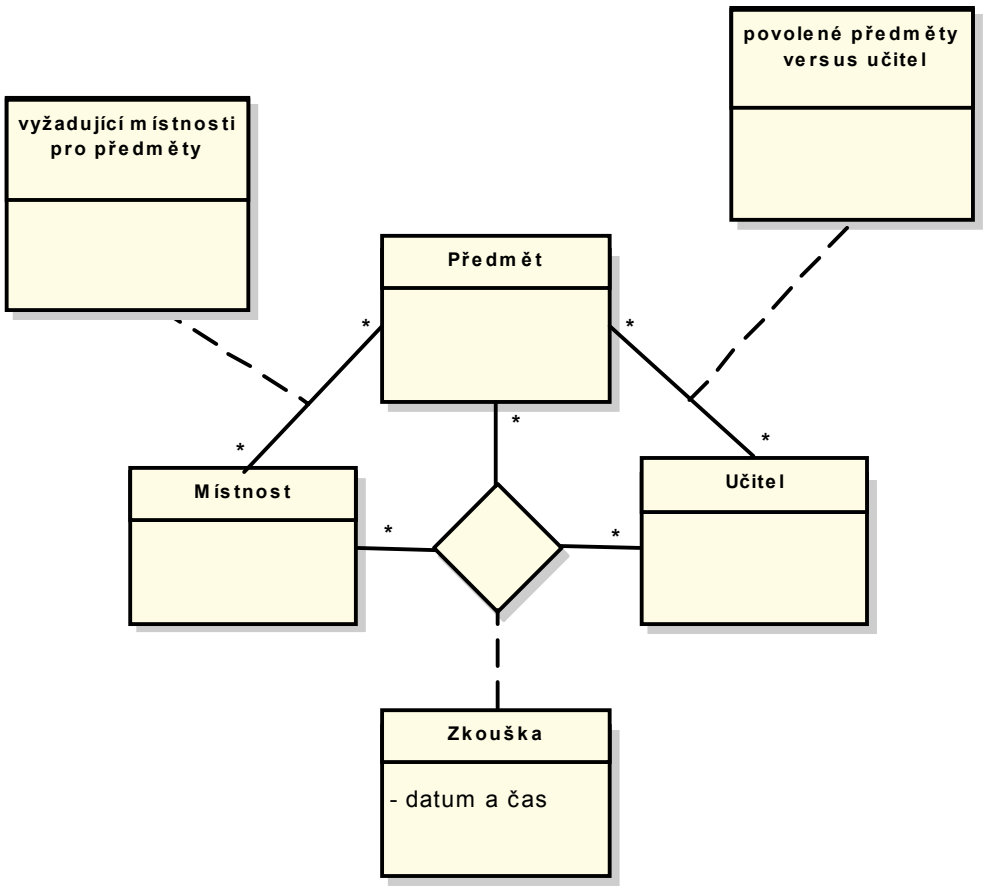
Dalším požadavek od konzultanta může znít například takto: „Ne každý učitel může zkoušet libovolný předmět. Bylo by dobré, aby se tento výběr omezil a to tak, že když se vybere učitel, tak se nabídnou pouze ty předměty, které učitel může zkoušet a naopak, pokud se vybere předmět, tak se nabídnou pouze ti učitelé, kteří mohou daný předmět zkoušet.“

Tento požadavek nás jako analytiky vede k zavedení asociativní třídy mezi třídou Učitel a třídou Předmět, například takto:



obrázek 90 Další asociativní třída povolené předměty versus učitel

Podobně bychom vyřešili další požadavek: Ne každý předmět lze zkoušet v libovolné místnosti (zkoušku z fyziky nebudeme dělat na pitevně). Nabízí se podobné řešení jako asociativní třída mezi třídou Místnost a Předmět, avšak je třeba zvážit, co tato informace ponese. Pokud bychom evidovali „povolené kombinace“, mohlo by se stát, že bychom museli evidovat příliš mnoho instancí. V tomto případě musíme vyzpovídat konzultanta, aby nás seznámil s tím, které kombinace jsou nejčastější, které méně časté apod. Nechť jsme tedy v tomto příkladu zjistili, že nejjednodušší bude evidovat pomocí asociativní třídy kombinace mezi místností a předmětem jako „vyžadující“, tj. některé předměty vyžadují určité místnosti na zkoušku (pitevna, fyzikální laboratoř apod.). Pokud se instance vyskytuje, vyžaduje se ji použít, pokud se nevyskytuje ani jedna vyžadující kombinace pro daný předmět, tak se má za to, že může být vybrána libovolná místnost.

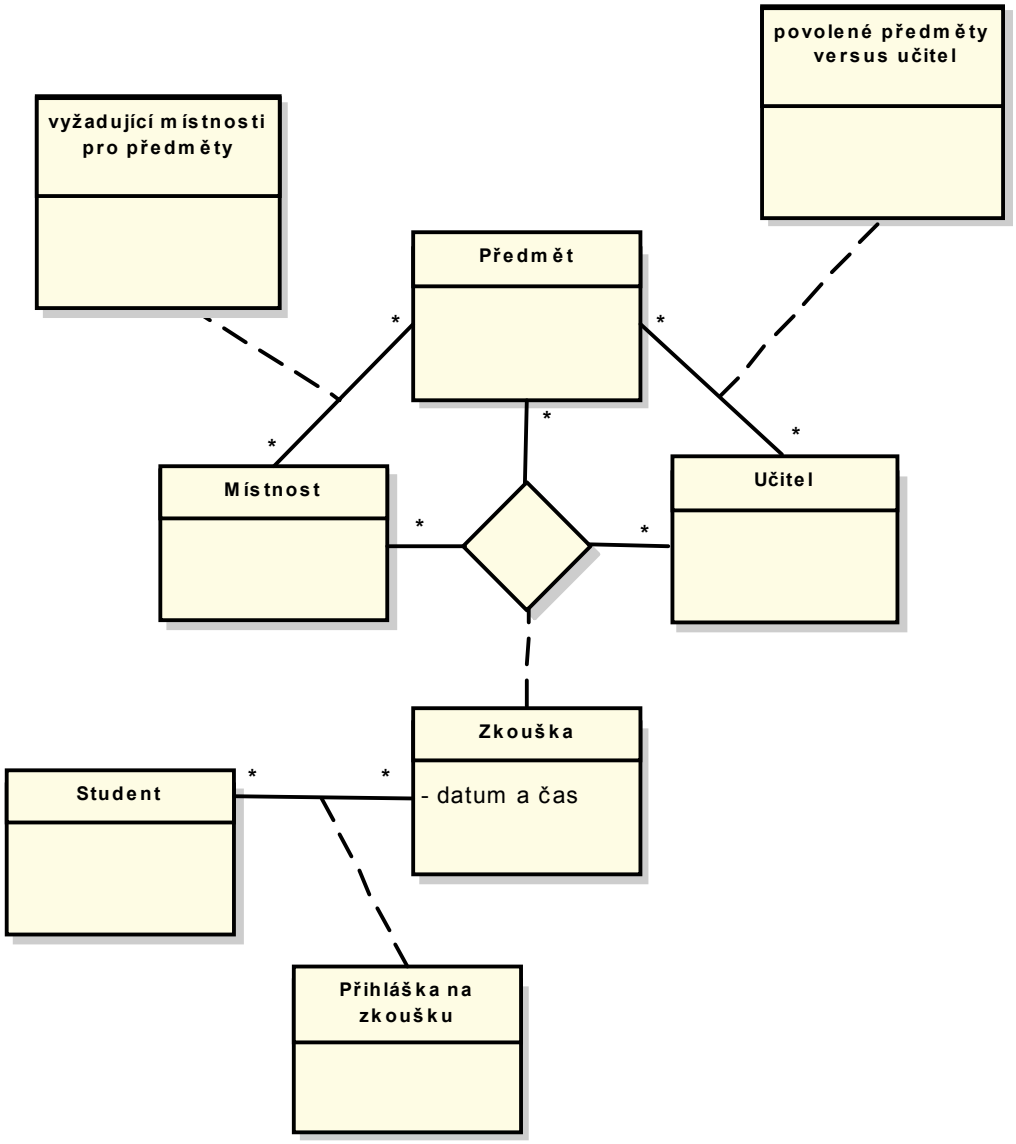


obrázek 91 Model doplněn o vyžadující místnosti versus předmět

Dalším požadavkem je evidovat studenty na zkoušce. Zde se nabízejí dvě takřka rovnocenné varianty analytického návrhu a je dobré si ukázat obě, abychom lépe pochopili jejich rozdíl.

V každém případě musíme zavést třídu Student. Když se daná obsluha přihlásí do systému, následně se identifikuje instance z této třídy.

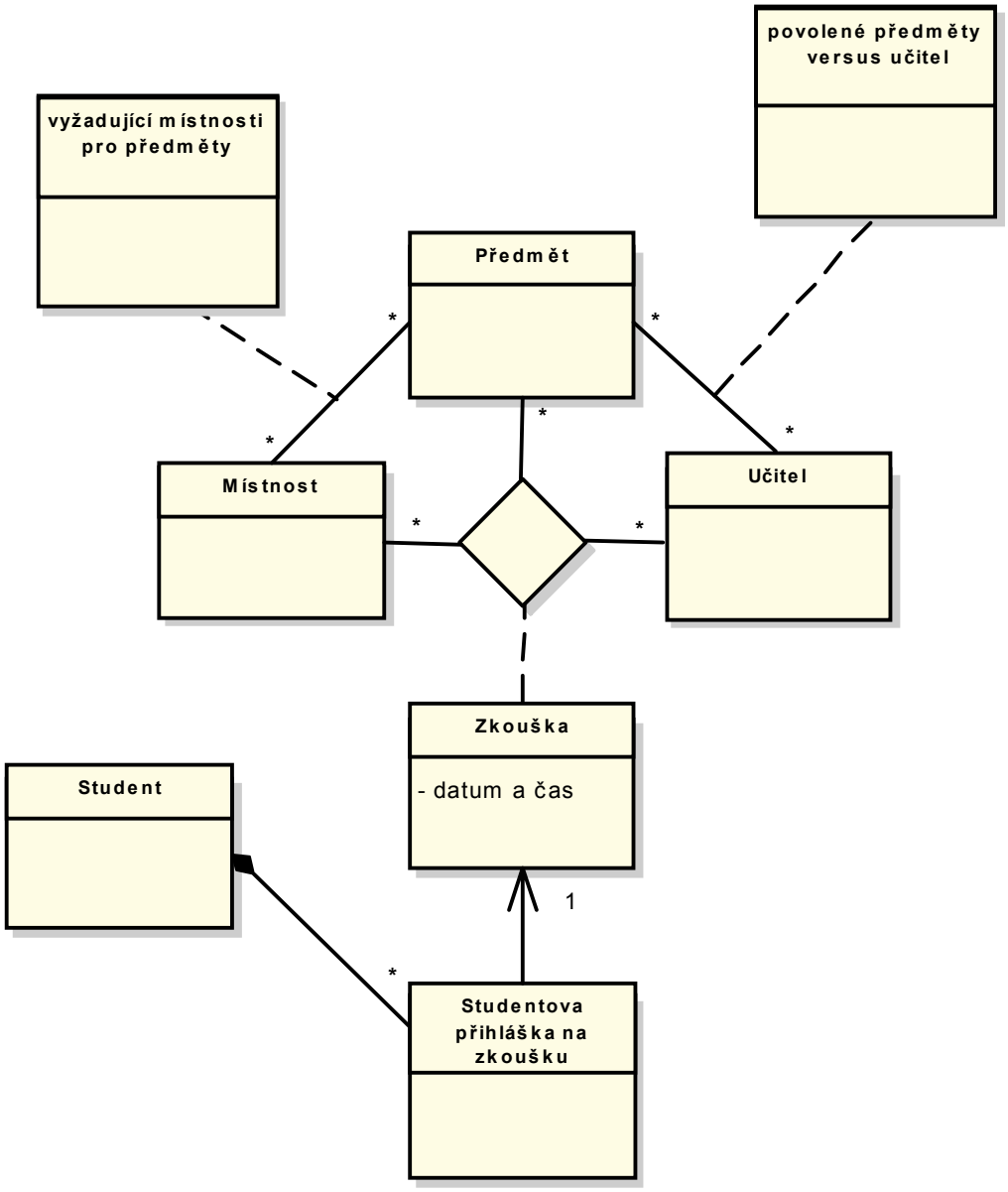
První varianta evidence studenta na zkoušce spočívá v použití asociativní třídy mezi studentem a zkouškou takto:



obrázek 92 Přidána třída Přihláška na zkoušku

V tomto případě se evidence chová takto: Existuje nezávislý seznam studentů a existuje nezávislý seznam zkoušek. K tomu navíc existuje seznam přihlášek na zkoušky, který tyto dva seznamy provazuje. Přidat přihlášku znamená mít vybraného studenta (přesněji instanci), mít vybranou zkoušku. Je třeba si uvědomit, že nezávislost entit Zkouška a Student vede k tomu, že tyto dvě části systému o sobě nevědí (navzájem se nepoužívají). Na druhou stranu při vymazání studenta ze seznamu studentů musí ten, kdo maže, vědět, že se musí mazat i jeho přihlášky. To platí obecně při práci s asociativní třídou: Na jedné straně vzájemná nezávislost, na straně druhé nutnost obsloužit i tento výmaz. Prakticky je vhodné tuto operaci vymazání přihlášek při výmazu studenta navrhnout pomocí vzoru OBSERVER z DESIGN PATTERNS, tj. nepřímým oslovením. Nejjednodušší řešení je umístit seznam přihlášek do seznamu „observerů“ pro událost v operaci vymazání studenta a tak zavolat vymazání všech přihlášek daného studenta. Vstupním parametrem této operace seznamu přihlášek musí být daný student nebo jeho identifikátor.

Druhá možnost by mohla vypadat takto:



obrázek 93 Druhé možné řešení přihlášek

Strukturou instancí je řešení velmi podobné předešlému. Například mapováním do relační databáze dostaneme úplně stejné datové struktury, tj. tabulek. Rozdíl je v tom, že v tomto řešení instance studenta „přímo znají“ a ovládají své přihlášky jako své děti v kompozici, tedy jako svůj seznam. Z hlediska kódu platí to, že ten, kdo bude držet instanci studenta v evidenci a bude ji chtít z evidence vymazat, tak zavoláním studenta k jeho vymazání tato instance sama vymaže své přihlášky přímým oslovením (zavolá „své děti“ anebo „seznam svých dětí“).

Toto druhé řešení má nevýhodu v tom, že třída Student je ve vztahu závislosti (DEPENDENCY) na třídě Studentova přihláška na zkoušku, protože jsou to její děti v kompozici. Stejně tak obráceně díky vztahu k parentovi je i obrácený vztah závislosti. Současně je ve vztahu závislosti (DEPENDENCY) třída Přihláška na třídě Zkouška. Vzniká tak poměrně velká provázanost kódu. V tomto případě by se asi nejednalo o příliš velký problém, ale pokud se příliš často používá takováto konstrukce, tak potom „každý musí znát každého“ a systém se stává obrovským nedělitelným molochem. Neměli bychom tyto vzájemné vztahy závislosti přehánět, protože potom se systém stává fyzicky nedělitelným.

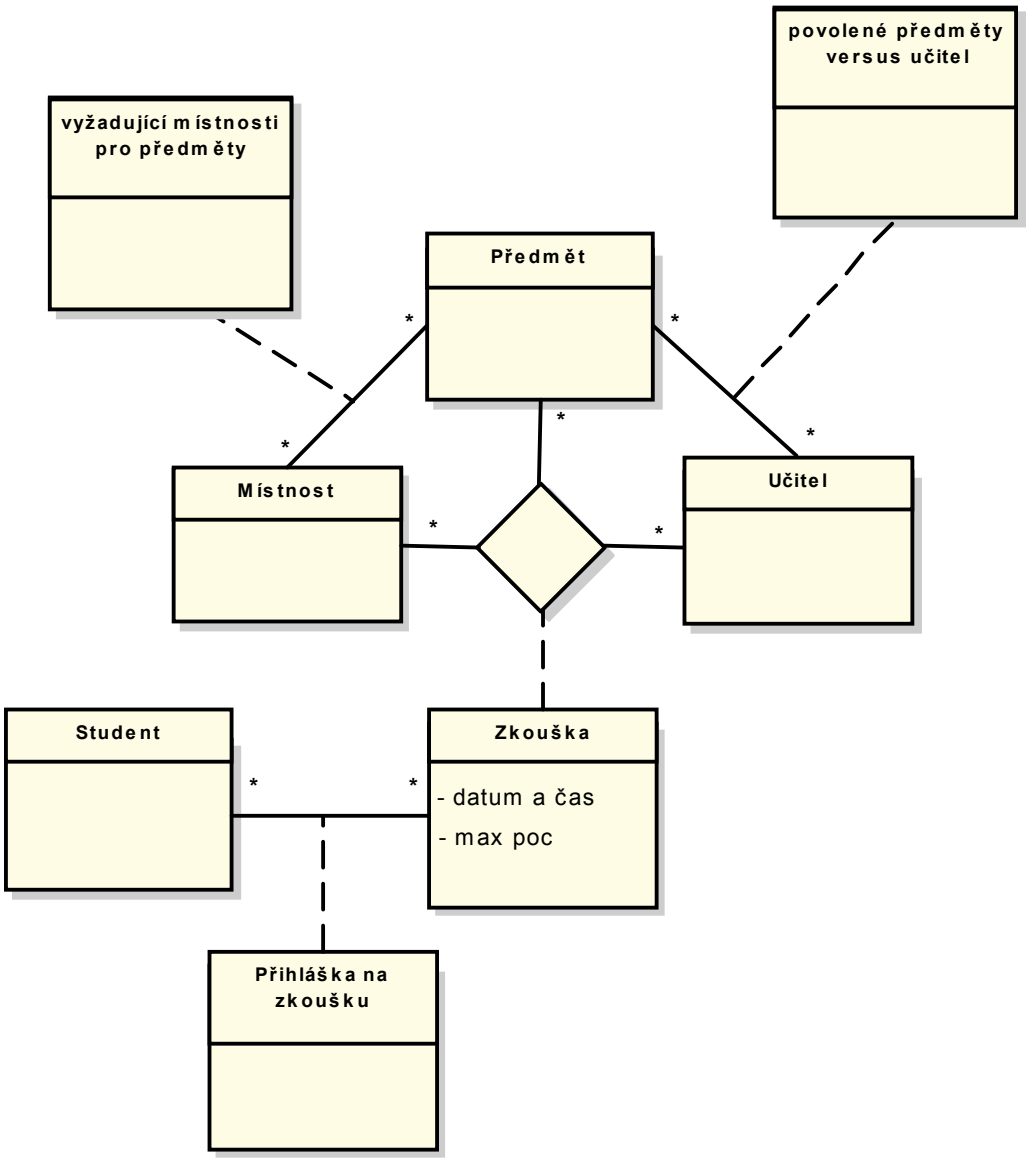
Pro náš příklad zvolíme první variantu s asociativní třídou a nikoliv řešení s kompozicí.

Další požadavek od konzultanta může znít takto: Ten, kdo pořádá danou zkoušku, bude chtít omezit kapacitu studentů na této zkoušce. Přece jen, když se systém spustí, tak se na danou zkoušku může přihlásit libovolný počet studentů a to zkoušející opravdu nechce. Otázka je, kam se tato informace „maximální kapacita studentů na zkoušce“ umístí a jak se s ní bude pracovat?

Několikrát se mi při školeních stalo, že někteří pracovníci navrhli umístit tuto informaci do instancí ze třídy Místnost. Jedná se přece o kapacitu místnosti, ne? Odpověď zní: Kapacita místnosti je zajímavá informace, ale tak formulace požadavku nezněla! Chce se kapacita zkoušky a ne kapacita místnosti!

V této chvíli je dobré dát jednu radu všem analytikům. Pokud se v analytickém modelování dobře formuluje otázka a dobře ji posloucháme, tak v jejím znění je skryta odpověď. Takže otázka zní: Kam patří maximální kapacita studentů na zkoušce? Pokud jsme si otázku dobře vyslechli, tak známe odpověď: Maximální kapacita zkoušky patří do instancí ze třídy Zkouška. Umístíme tedy do třídy nový atribut tohoto významu.

Maximální počet studentů na zkoušce tedy zavedeme jako atribut zkoušky, zde jako s názvem „max poc“:



obrázek 94 Doplněn atribut max poc

Než navrhneme analytickou funkcionalitu, jak se s touto informací „maximální počet studentů na zkoušce“ bude systém pracovat, podívejme se na předešlý obrázek a zeptejme se: Může se student přihlásit dvakrát na tutéž zkoušku a mít tam tedy dvě přihlášky? I když to zní nyní podivně, tak odpověď zní: NE. Je to dáno tím, že v našem modelu evidence zkouška není zkouškou v původním slova významu, ale odpovídá tomu, co se jinak nazývá termín zkoušky s přesným datumem časem. Z toho důvodu se v přihláškách budou vyskytovat unikátně výskyty kombinace výskytů student a zkouška. Z toho důvodu můžeme přejmenovat třídu Přihláška na

jiný výstižnější název Student na zkoušce, protože už to nemusí být pouze přihláška, můžeme u těchto instancí evidovat všechny možné údaje ohledně studenta na zkoušce, například kdy se přihlásil, jak zkouška dopadla apod.

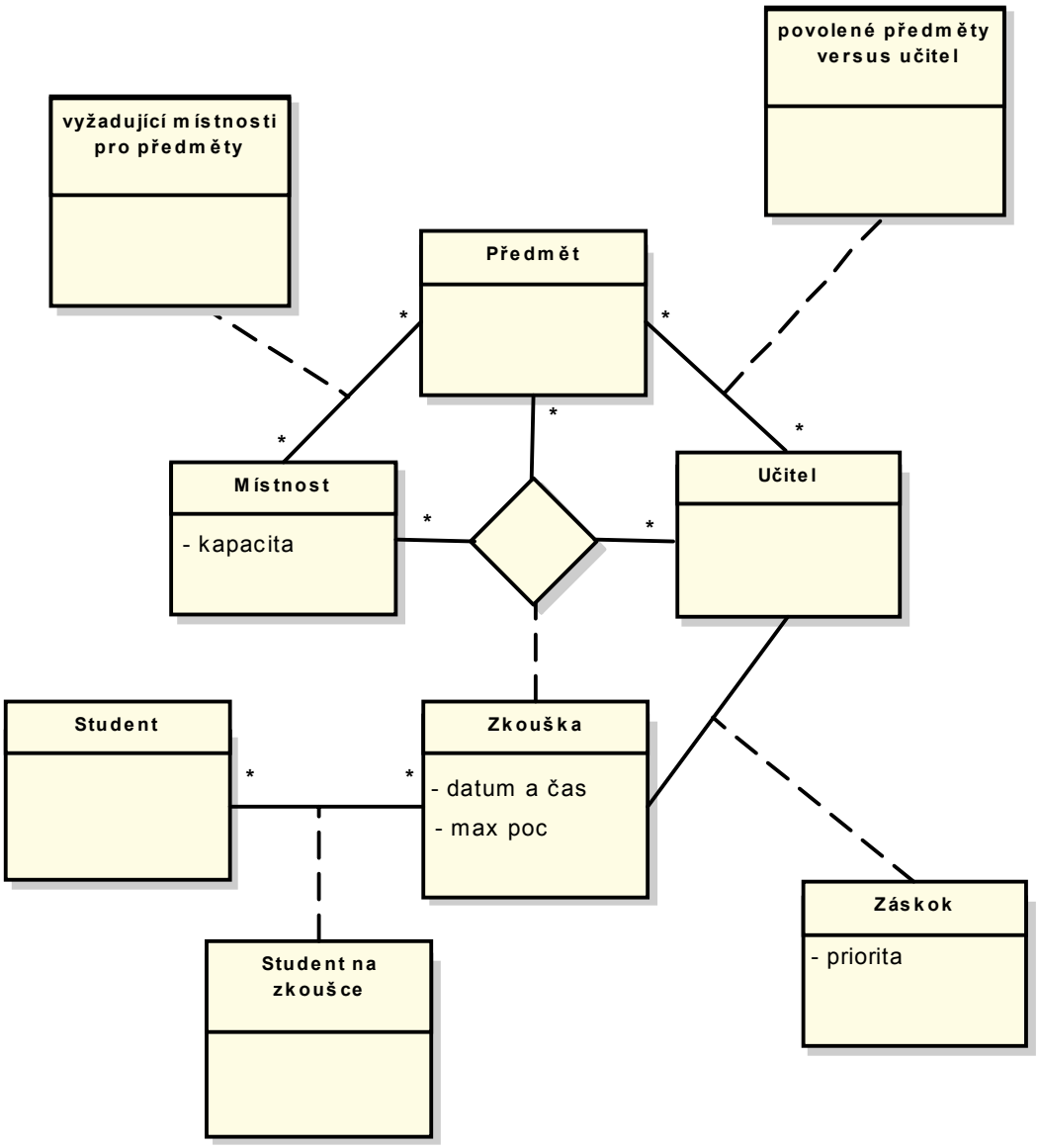
Můžeme navrhnout tuto funkcionalitu (ta není součástí modelu tříd, ale některého z dynamických modelů, např. USE CASE MODELU), která využívá hodnotu v tomto atributu: Studentovi se zobrazí zkoušky, ke kterým se může přihlásit. Ty zkoušky, které mají hodnotu v atributu „max poc“ větší nebo rovnu počtu instancí studentů na zkoušce pro danou zkoušku, se tzv. vyšedí, jsou neaktivní a nelze se na ně přihlásit.

Můžeme zavést i nepovinně vyplňovaný atribut kapacita místnosti. Tato hodnota má informativní charakter a obsluha ji může ignorovat. Může se využít tak například tak, že pokud obsluha zadává hodnotu maximálního počtu studentů na zkoušce větší, než je kapacita místnosti, systém na to obsluhu upozorní, ale případně ji pustí dále.

Příklad však ještě nekončí. Konzultant vysloví tento požadavek: „Někdy se stane, že daný určený učitel, který mý provádět danou zkoušku, tzv. ‚vybouchne‘. Potřebovali bychom evidovat záskok učitelů na zkoušce, může jich být i více.“

Na školení se někdy posluchači dopustili té chyby, že tento požadavek navrhli realizovat asociativní třídou mezi učiteli (učitel na učitele), což není přesné. Vlastně touto chybou vzniká zajímavá otázka: Co by mohla vyjadřovat vazba učitel versus učitel ve smyslu záskoku? Někteří učitelé si přejí, aby jej případně přednostně zaskočil některý z kolegů. Evidence „učitelé přednostně zaskakující k danému učiteli“ by byla realizována právě asociativní třídou „učitel - učitel“ a využila by se tak, že by se obsluze nabízela jako „přednostní seznam učitelů“ pro zadání samotného záskoku, který zatím nemáme vyřešen.

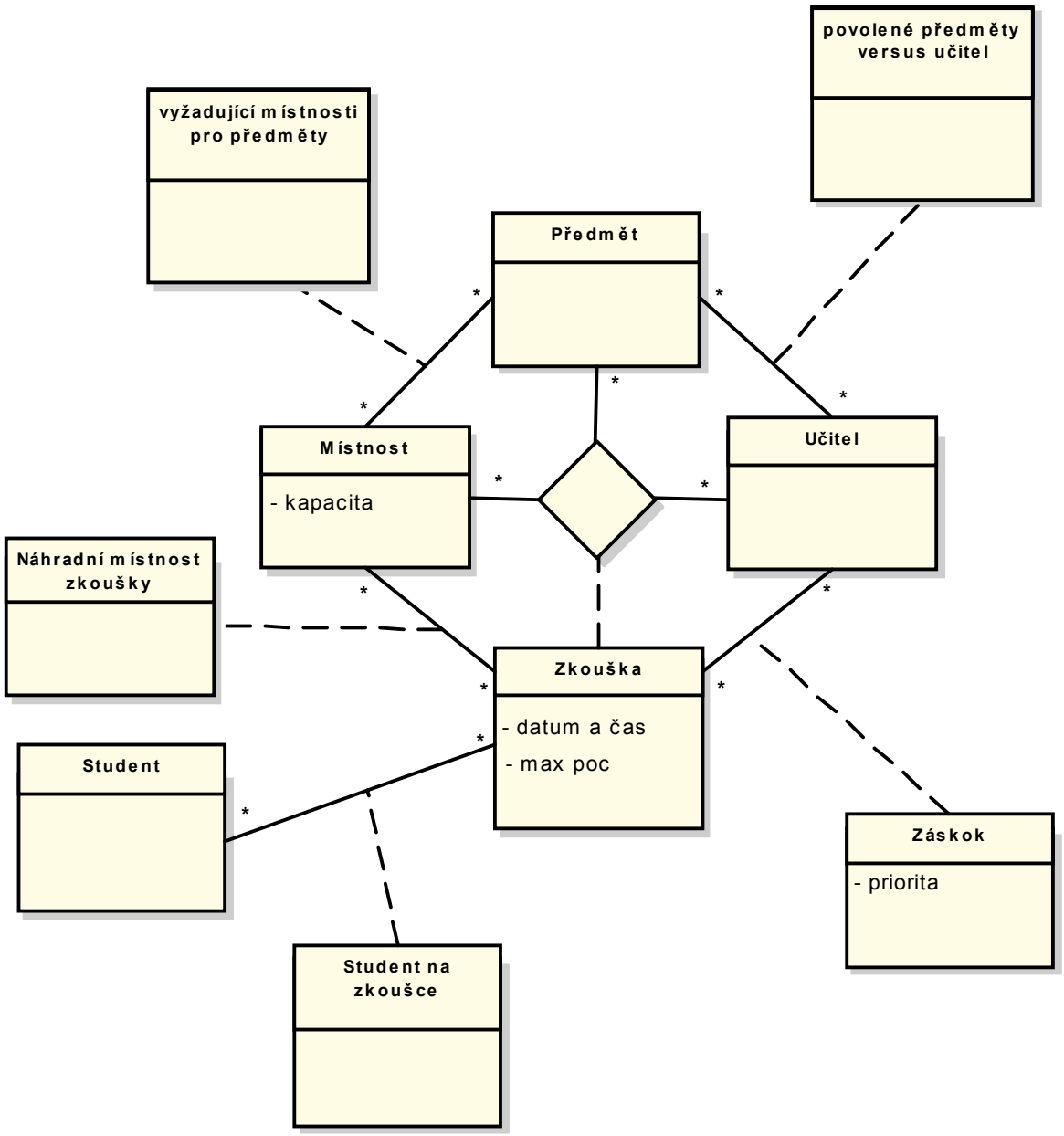
Připomeneme si zásadu: Když dobře posloucháme, známe odpověď. Zadání zní takto: Záskok učitele na zkoušce. To vede k asociativní třídě mezi zkouškou a učitelem. Navíc ještě vybavíme instance prioritou, která pro danou instanci zkoušky udává pořadí, např. takto:



obrázek 95 Doplněno o záskok učitele na zkoušce

Všimněme si, že řešení záskoku umožňuje přehodnotit celý model a zvolit druhou variantu. Pokud bychom záskok zobecnili i do té polohy, že první záskok na zkoušce určuje toho, kdo má učinit zkoušku (něco jako „nultý“ záskok), odpadla by nám jedna přímá vazba mezi zkouškou a učitelem v asociativní třídě zkouška (jeden její konec směrující na učitele), to by ale již nebyl příklad na násobnou asociativní třídu. Je otázkou, jestli toto řešení zvolit, každé pro má své proti.

Podobně bychom mohli zavést další „záskok“ a tím by bylo určení náhradní místnosti pro zkoušku. Analogické řešení s asociativní třídou náhradní místnosti:



obrázek 96 Doplněno o náhradní místnost zkoušky

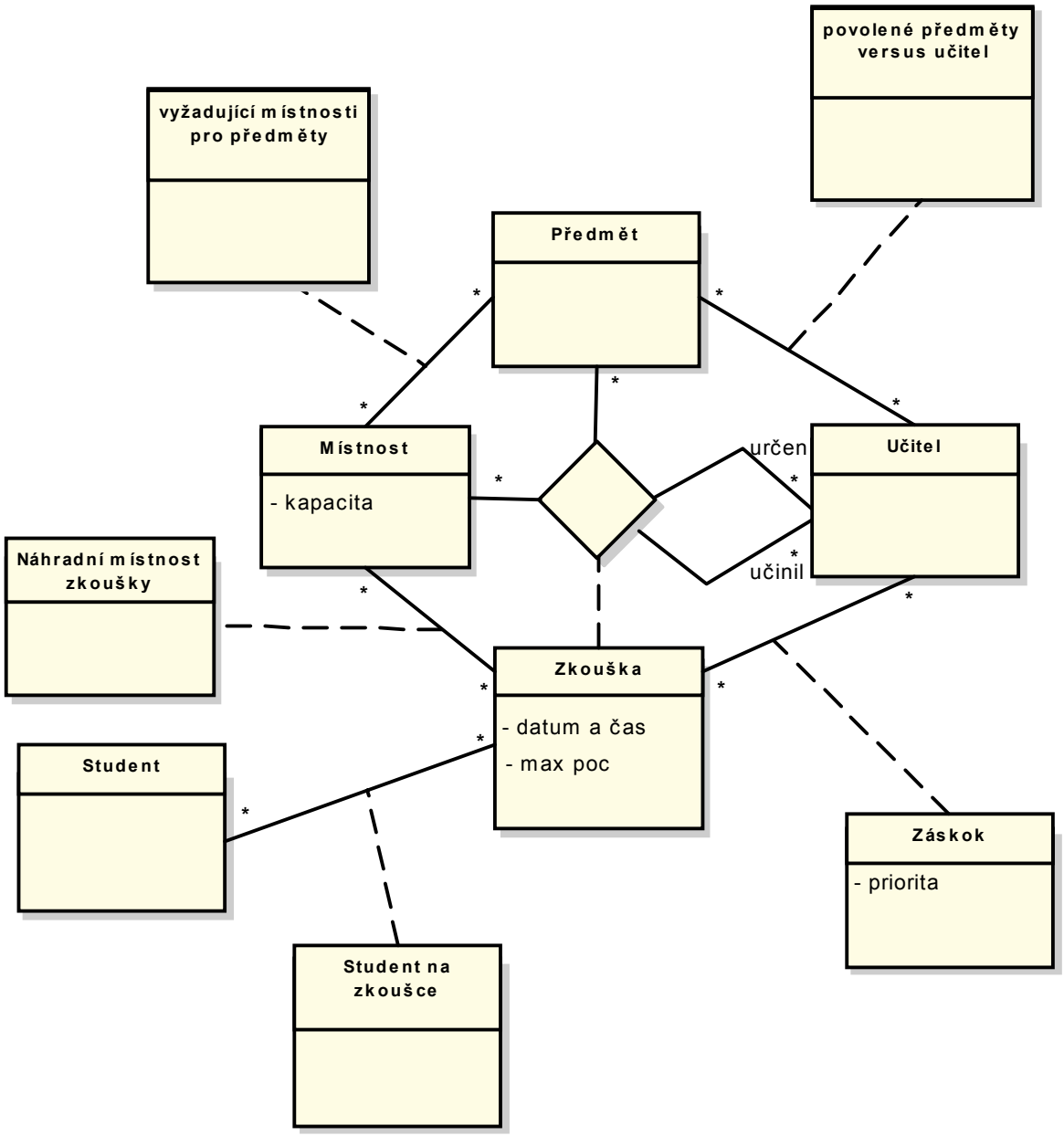
Podobně by bylo možné první náhradní místnost prohlásit za tu místnost, kde se má původně zkouška odehrát a odpadla by další přímá vazba (konec na místnost) v asociativní třídě Zkouška.

Samozřejmě, záskok předmětu na zkoušce dělat nebudeme, protože to by se studentům asi moc nelíbilo („…dnes se nezkouší z lineární algebry, ale máme náhradní předmět zkoušky z matematické analýzy…“ ☺).

Jako poslední evidenci v tomto příkladu bychom mohli sami jako analytici nabídnout konzultantovi určitou funkcionalitu systému, která tam nyní není, ale určitě by uživatelům chyběla a za pár týdnů by se ozvali. Jakmile jsme totiž připustili možnost záskoku učitele na zkoušce, tak tím není v evidenci apriori dáno, že určený učitel je ten, kdo ji skutečně učinil. Znamená to, že budeme chtít evidovat i učitele na zkoušce, který ji opravdu udělal. Samozřejmě většinou to bude ten, kdo je určen, ale až se budou rozdělovat prémie za odvedené zkoušky, průkazná evidence bude opravdu hodně zapotřebí.

Existují dvě možnosti, jak tuto evidenci „který učitel vedl tuto zkoušku“ navrhnout. V obou případech to bude znamenat, že z instance zkoušky povede nový ukazatel na danou instanci učitele ze seznamu učitelů.

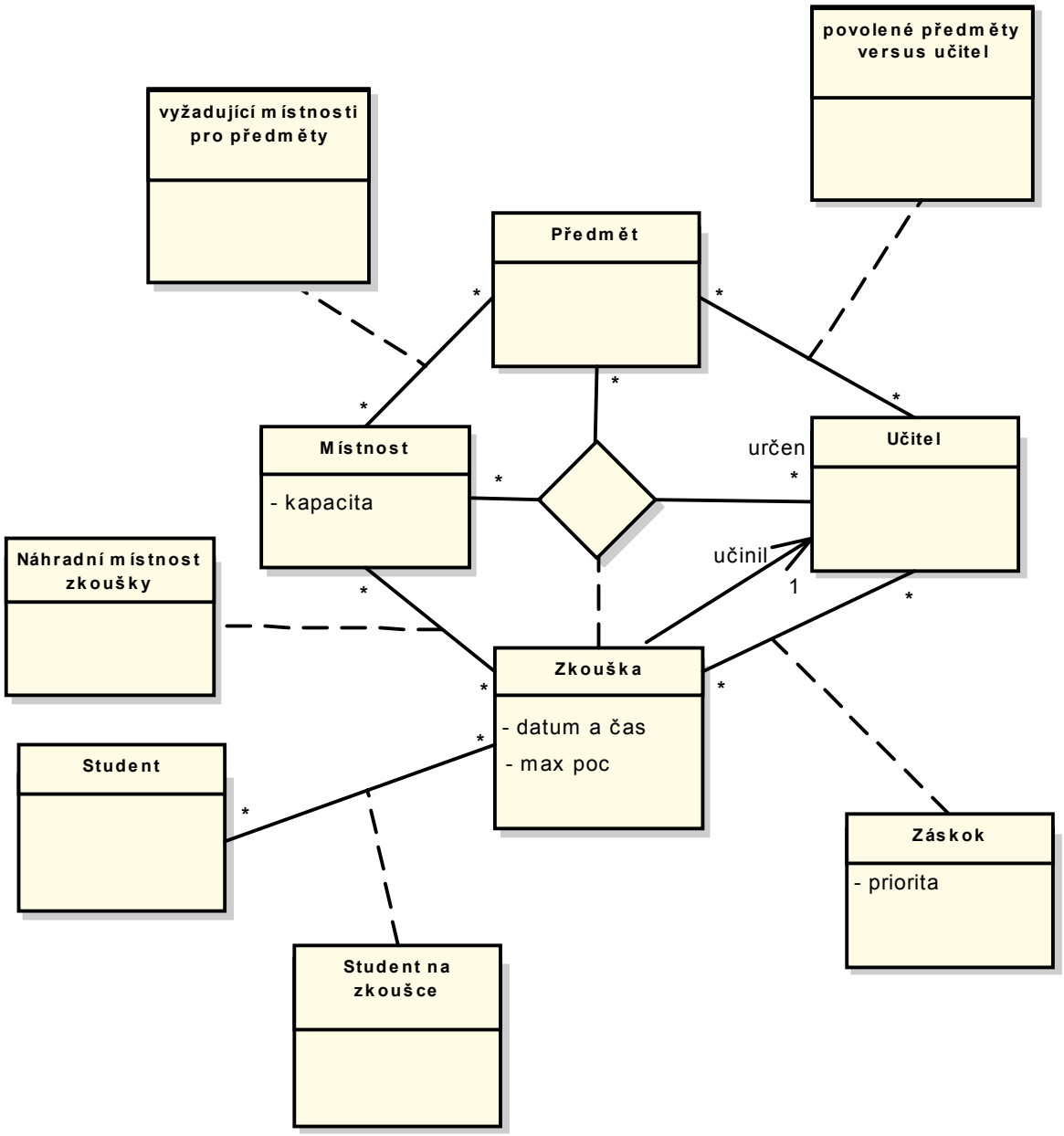
Nabízí se zajímavá možnost z násobné asociativní třídy udělat ještě více násobnou třídu, tj. vyvést ještě jednu vazbu na učitele. Nyní již však určitě musíme zavést role, aby bylo jasné, proč ze zkoušky vedou další vazby:



obrázek 97 Zkouška ještě více násobná třída s navíc jedním koncem na učitele

V tomto modelu si bude instance zkoušky „ukazovat“ na dvě instance v seznamu učitelů, jednou z důvodu (tj. role) „kdo je určen“, po druhé z důvodu (role) „kdo ji učinil“.

Následující řešení je velmi podobné:



obrázek 98 Jiné řešení s evidencí učitele, který učinil zkoušku

I v tomto případě zkouška v evidenci „vidí“ instanci učitele v roli „kdo ji učinil“. Rozdíl oproti předešlému návrhu je v tom, že tato evidence „kdo ji učinil“ se neúčastní celé té hry mezi konci násobné asociativní třídy ve smyslu: Když vyberu předmět, vidím zkoušky a učitele, kteří jsou určeni a místnosti, které jsou určeny atd.

Osobně bych se raději přiklonil k druhému řešení s číselníkovou vazbou, ale musím zdůraznit, že rozdíl je minimální a určitě čas strávený diskusemi nad těmito dvěma řešeními neodpovídá důležitosti tohoto problému.

4.40 Vztah GEN SPEC a vztah ASSOCIATION v UML

Vztah GENERALISATION - SPECIALISATION, neboli generalizace a specializace, (budeme dále také označovat jako GEN-SPEC) se podstatně liší od všech vztahů, které jsme zatím probrali.

Všechny předešlé vztahy měly jednu vlastnost společnou: Zaváděly vztahy mezi instancemi, tj. ukazovaly, jak jedny instance používají jiné instance. V UML všechny tyto vztahy spadají pod jednu velkou oblast a nazývají se obecně ASSOCIATION, neboli asociace (pozor, my jsme doposud znali v analytickém modelování běžnou asociaci pouze jako její zvláštní případ!).

Zapamatujme si základní vlastnost asociace: Asociace (ASSOCIATION) dává do vztahu instance informace, tj. jak instance používají jiné instance.

Vztah GEN-SPEC patří do úplně jiné samostatně stojící skupiny vztahů. Jedná se totiž o vztah na vyšší úrovni meta mezi třídami, tj. jedná se o přímý vztah použití třídy třídou, tj. vztah GEN-SPEC zavádí použití jedné třídy druhou třídou. Zatímco asociace zavádí použití mezi instancemi, tj. jak se mezi nimi přechází, jaké mají vazby atd., tak GEN-SPEC udává použití jedné třídy druhou třídou jako mezi typy.

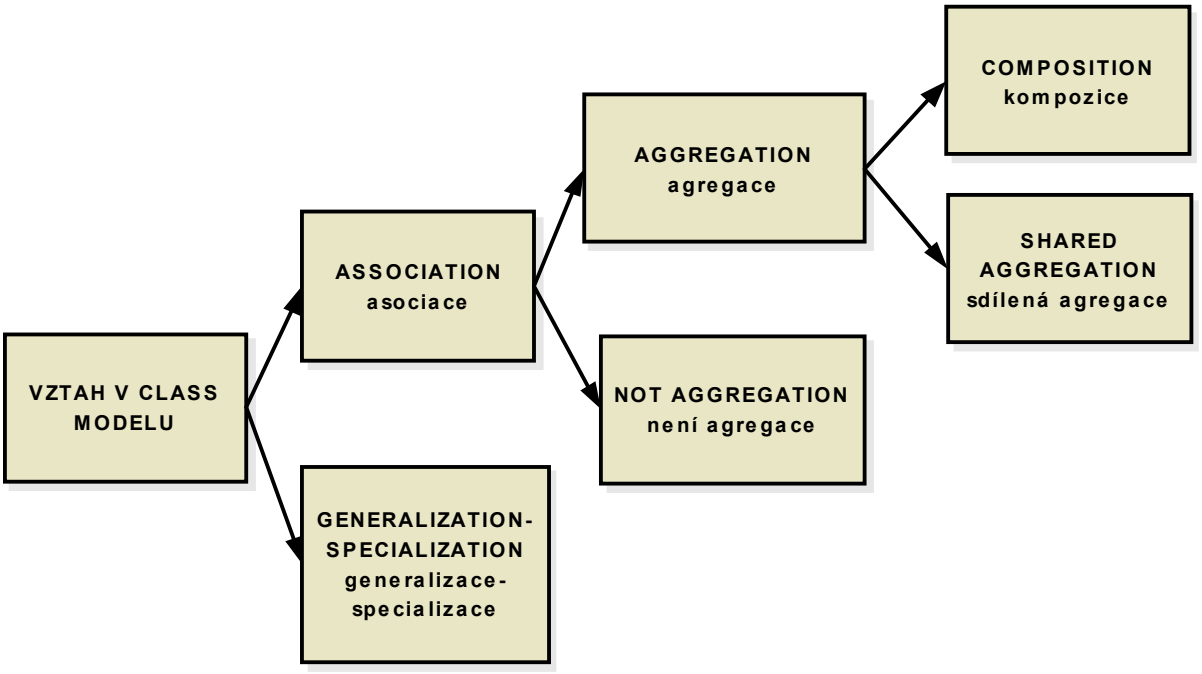
Můžeme si uvést následující přirovnání: Představme si, že třídy jsou stroje na boty a instance jsou boty. Vztah asociace udává, jak budou jednotlivé boty, které vyplivnou stroje, mezi sebou sešněrovány. Vztah GEN-SPEC udává něco jiného: Jedná se o interakci mezi samotnými stroji, kdy jeden stroj nějak použije (ještě před spuštěním výroby bot) druhý stroj, stroje se poskládají a poté stroj vyplivne boty s vlastnostmi, které jim dalo použití jednoho stroje druhým strojem.

Dá se říci, že zatímco asociace udává, jak se skládají instance, vztah GEN-SPEC udává, jak se skládají třídy na meta úrovni.

4.41 Syntaxe UML a vztahy v modelu tříd

Smyslem této kapitoly je uvést pozici vztahu GEN-SPEC v UML vůči ostatním vztahům, které jsme již probrali a současně dát dohromady předešlý výklad týkající se pohledu na systém pomocí analytického modelu tříd a správné a čisté syntaxe UML.

Základní rozdělení vztahů v CLASS MODELU UML ukazuje následující obrázek:



obrázek 99 Rozdělení vztahů v CLASS MODELU UML

Vztah v CLASS MODELU je buď asociací (vede ke vztahu mezi instancemi) anebo vztahem GEN-SPEC (vztah použití třídy druhou třídou).

Pokud je vztah asociací, tak je buď agregací (vztah celek část a existuje kosočtverec na jednom z konců) anebo není agregací (není žádný kosočtverec).

Pokud je agregací, tak je buď kompozicí (vztah plného majitelství, černě vyplněný kosočtverec), anebo je sdílenou agregací (vztah sdíleného majitelství, nevyplněný kosočtverec).

Syntaxe UML dále zavádí asociativní třídu velmi jednoduše: Jedná se o asociaci, která je současně i třídou (tj. třídou, jejíž instance zprostředkovávají vztah mezi instancemi). Nutno podotknout, že tato definice je sice přesná a hutná, ale pro neznalého absolutně nepochopitelná. Naštěstí principy fungování asociativní třídy máme vysvětleny (viz příklad rodokmen) a proto nás tato „hutná“ definice nezaskočí.

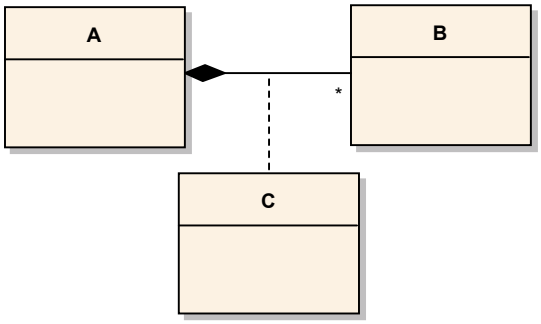
Spolu s další syntaxí UML, která se zavádí u konců asociace, jako jsou „IsNavigable“ (prostupnost, směrovost vazby), multiplicita (násobnost) a role, můžeme pomocí předešlé syntaxe vztahů vyjádřit všechny vztahy, které jsme probrali v analytickém modelu tříd.

Vyjmenujeme si již probrané vztahy analytického modelu v kontextu s předešlým obrázkem vztahů v UML:

1. Analytik hledá kompozici (nebo sdílenou agregaci) od majitele ke svým částem, vztah buď ku 1 nebo ku N, tj. případ, kdy „instance vlastní instanci“: To odpovídá v syntaxi UML kompozici nebo sdílené agregace v daném směru od celku směrem k částem (neuvažuje se v té chvíli opačný směr, tj. je třeba se dopátrat zda bude ve vztahu šipka ano nebo ne).
2. Běžná asociace jako směrový vztah, tj. případ, kdy „instance používá druhou instanci, ale nevlastní ji“. Analytik rozhoduje mezi dvěma možnostmi: Buď se jedná o vztah asociace bez označení agregace ve směru od jednoho prvku k jednomu druhému („číselníková vazba“) anebo se jedná o „vztah k parentovi“, což je obrácený vztah v agregaci od jednoho prvku z N částí k celku.

- Ostatní syntaxe, tj. asociativní třída a GEN-SPEC, odpovídají členění názvů podle UML.

Ještě jednu poznámku je třeba učinit: Asociativní třída je z hlediska syntaxe UML chápána jako vztah ASSOCIATION (vztah mezi instancemi), který je současně i třídou. Vyplývá z toho, že existuje i možnost zavést kombinaci „kompozice s asociativní třídou“, tj. na jednom konci by se objevil černý kosočtverec, který jsme ještě nebrali:



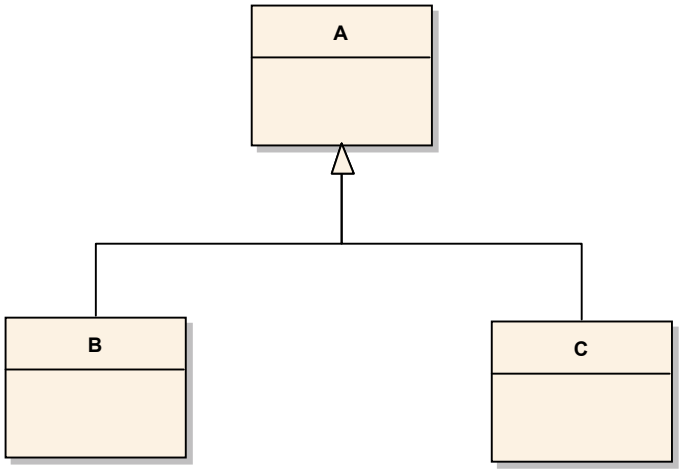
obrázek 100 Asociativní třída s kompozicí je povolena

V tom případě se daný vztah chápe v podstatě stejně, jako jsme již vysvětlili u „klasické“ asociativní třídy (třída C propojuje A s B), jenom existuje jediný podstatný rozdíl: S životem té instance, která má u sebe černý kosočtverec (v našem případě instance ze třídy A) se odvíjí život instancí jak ze třídy C, tak ze třídy B jako jejích částí. Například při vymazání instance A se vymažou jak „její“ instance z C, tak také „její“ instance z B. U příkladu s vlastnictvím aut tento vztah s kompozicí nepřichází v úvahu, protože seznamy aut a osob se chápou jako dva nezávislé seznamy: Při vymazání auta se žádná osoba v seznamu nemaže, resp. při vymazání osoby se tato operace aut také netýká.

4.42 Struktura instancí vzniklých pomocí GEN-SPEC

Vraťme se k výkladu ohledně vztahu GEN-SPEC. Vztah GEN-SPEC je směrovým vztahem a vyjadřuje opětovnou použitelnost. Na rozdíl od asociace se nepromítá do vztahu instancí, ale jedná se o interakci mezi třídami, typy, druhy. Jedná se tedy o „typovou interakci“.

Jedna třída může použít druhou třídu a tak provést tak zvanou specializaci. Použitá třída se nazývá generalizující (obecnější) a ta, která používá tuto třídu, se nazývá specializací. Interakce se vyjadřuje spojnicí s trojúhelníkem s vrcholem ukazujícím směr interakce použití ke generalizující třídě:

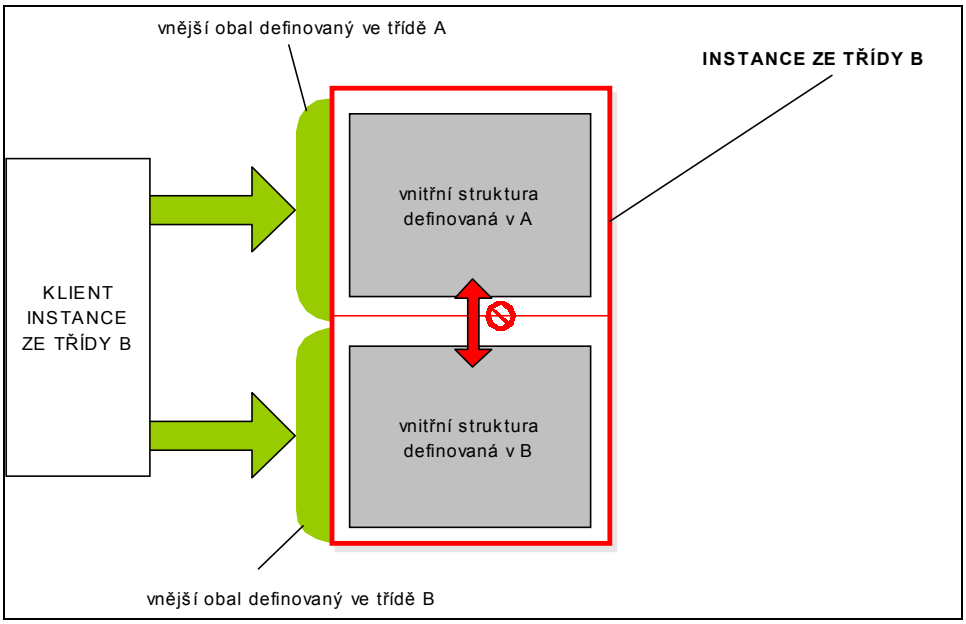


obrázek 101 Vztah GEN-SPEC, kdy třída B používá třídu A a také třída C používá třídu A

Na předešlém obrázku třída B a třída C používají třídu A. Třída A je obecnější, třída B a třída C jsou její specializace.

Je třeba blíže vysvětlit, jak vlastně funguje vztah na předešlém obrázku. Vztah GEN-SPEC je vztah použití mezi druhy a druhy jsou vlastnosti dávané instancím. Otázkou je, jak vypadá (například) instance ze třídy B na předešlém obrázku (samozřejmě analogicky jak vypadá instance ze třídy C).

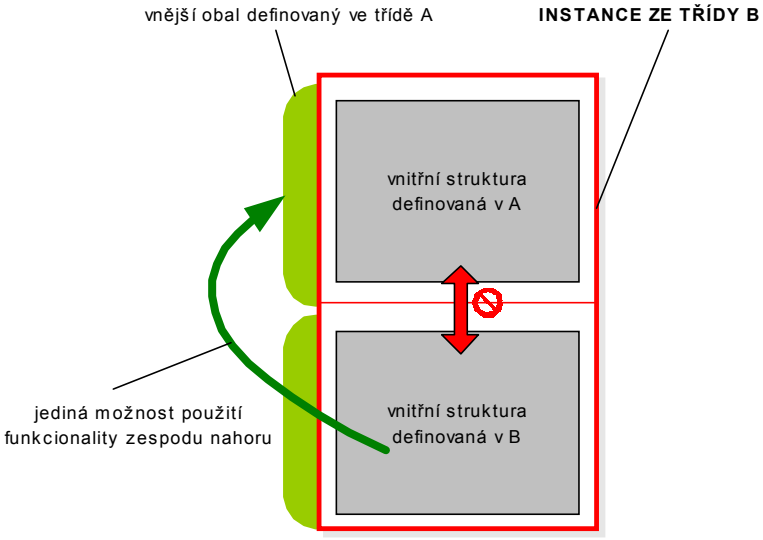
Strukturu instance B zobrazuje následující obrázek:



obrázek 102 Struktura instance při realizaci vztahu GEN-SPEC

Všimněme si, že instance ze třídy B obsahuje jak vnitřní strukturu definovanou ve třídě A, tak vnitřní strukturu definovanou ve třídě B. Tyto dvě struktury nejsou ve vzájemné viditelnosti (pozn.: v „čistém“ objektovém pojetí). Instance podporuje oba vnější obaly, jak vnější obal definovaný ve třídě A, tak ve třídě B. Klient instance ze třídy B může použít oba vnější obaly.

Zajímavá situace nastane, pokud se ve vnitřní struktuře definované ve „spodní speciálnější třídě“ B potřebuje použít něco, co je definované v „horní obecnější třídě“ A. I když nejsou vnitřní struktury v přímé viditelnosti, je to možné, ale existuje jediná možnost, jak toto učinit: Přes vnější obal definovaný v „horní“ třídě A. Obecně totiž platí, že každá instance poskytuje vnější obal pro užití libovolnému klientovi, který ji bude „držet“. Proto každá instance může poslat požadavek sama sobě, protože automaticky vždy drží ukazatel sama na sebe. Znamená to, že instance má vždy k dispozici svůj vlastní vnější obal a to i v případě toho obalu, který je definován v „horní“ třídě. Pokud tedy nějaká funkcionalita „dole“ potřebuje cokoliv „shora“, posílá požadavek vnějšímu obalu definovaném nahoře, graficky znázorněné např. takto:



obrázek 103 Jediná možnost, jak použít funkcionalitu v GEN SPEC zespodu nahoru

Poznámka: Některé jazyky umožňují v designu OOP opustit striktní požadavek „neviditelnosti“ horní vnitřní struktury ze strany spodní struktury (na předešlém obrázku se vypne červená dvojšipka se zákazem). Použije se například viditelnost typu **`protected`**. Tato konstrukce urychluje kódování, na straně druhé zvyšuje se riziko závislosti spodních struktur na vnitřních horních strukturách a systém se stává náchylnější k nestabilitám při změnách - spodek „vidí“ až do vrchní kuchyně. Důsledkem toho je, že pokud se „překope vrch“, musejí se vždy „překopat“ všichni dědicové.

Pokud se podíváme na obrázek 102, tak je z něj zřejmé, že instance ze třídy B bude mít ty vlastnosti, které jsou definované ve třídě B a ty vlastnosti, které jsou definované také v A. Tato vlastnost je evidentně transitivní, tj. pokud je A specializací jiné třídy, třída B přebírá transitivně také tyto vlastnosti ještě obecnější třídy, která by byla nad A (na obrázku není tato obecnější třída znázorněna).

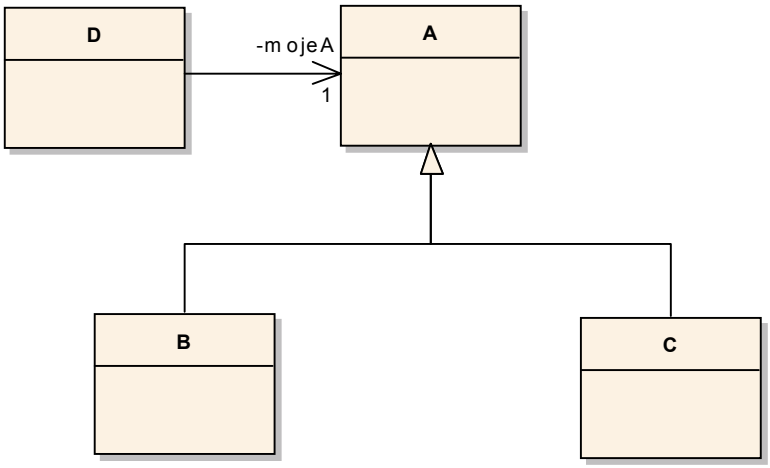
O vztahu GEN-SPEC se někdy také hovoří jako o „dědění“, což je pojem původně speciálně zaveden v OOP ve fázi D. Důvod je ten, že GEN-SPEC bývá v OOP mapován na dědičnost (inheritanci).

4.43 Zástupnost rolí v GEN-SPEC

Protože instance ze třídy B získá děděním v GEN-SPEC vlastnosti ze třídy A, tak všude tam, kde třída A vystupuje na konci asociace v nějaké roli, může hrát tuto roli také instance ze třídy B. Je to dáno tím, že také instance ze třídy B podporují vnější obal, který je definován ve třídě A.

Po mapování do designu se v OOP ve statických jazycích tato vlastnost projevuje tzv. „kompatibilitou typu, tj. tříd, zespodu nahoru“.

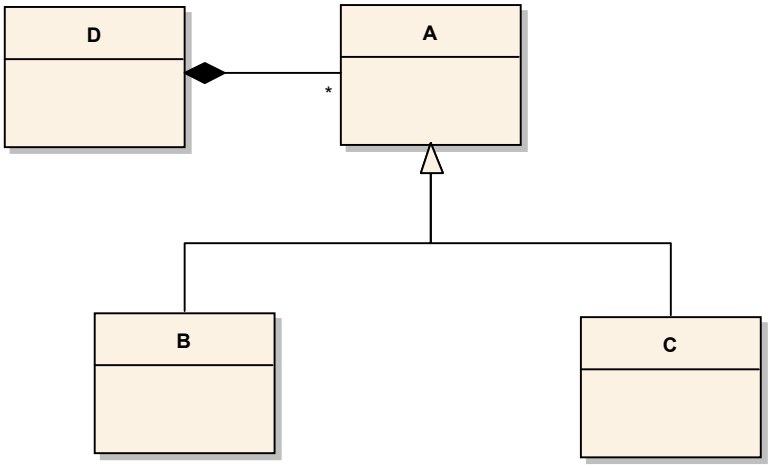
Tato skutečnost vede ke analytickým řešením, kdy na daném místě v dané roli může vystupovat několik podtypů daného typu, například takto:



obrázek 104 Příklad na zástupnost rolí

Na předešlém obrázku vidíme vztah asociace, konkrétně „číselníkovou vazbu“ od třídy D ke třídě A. Díky uvedené vlastnosti zástupnosti rolí může teoreticky na místě role „mojeA“ na straně asociace u třídy A hrát roli instance jak ze třídy A, tak instance ze třídy B, stejně tak instance ze třídy C. Je to dáno tím, že instance všech tří typů podporují tentýž obal (tentýž interface) definovaný ve třídě A. Je důležité si uvědomit, že ze strany klienta této role, tj. ze strany od D, je tato instance viděna jako instance typu A, ale v ní může být dosazena instance buď z A, nebo z B nebo z C.

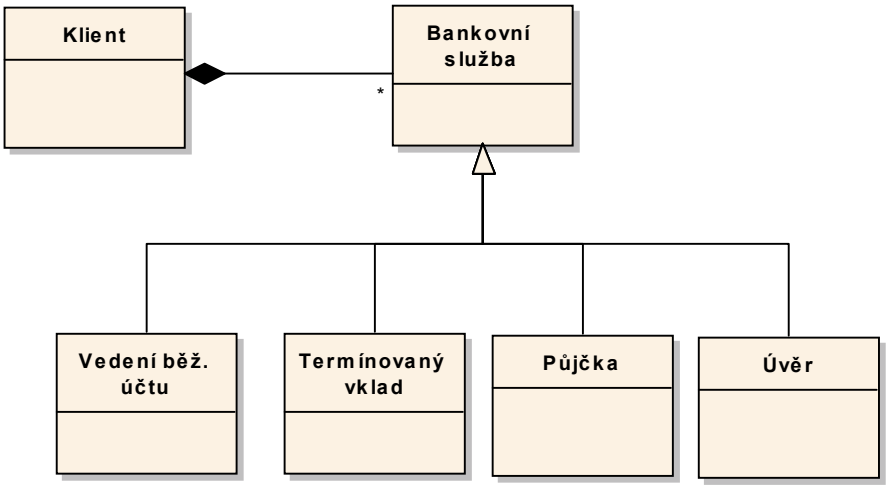
Podobně může vzniknout „heterogenní seznam“ takto:



obrázek 105 Jiný příklad na zástupnost rolí

V tomto případě se jedná o asociaci typu kompozice. V seznamu kompozice na straně u třídy A se mohou vyskytovat instance jak ze třídy A, tak ze třídy B, tak ze třídy C. Jinak řečeno, vznikne heterogenní seznam, ve kterém jsou prvky různých typů. Majitel těchto instancí, tj. držitel na straně u třídy D, je všechny používá jako instance typu A, ale v seznamu jsou prvky různých podtypů.

Jako příklad zástupnosti rolí si uvedeme následující diagram:



obrázek 106 Zástupnost rolí v kompozici

Všimněme si na předešlém obrázku, že Klient drží seznam Bankovních služeb a v těchto instancích seznamu se mohou vyskytovat instance jak ze třídy Vedení běžného účtu, tak instance z typu Termínovaný vklad, tak ze třídy Půjčka, tak ze třídy Úvěr. Pro představu, v seznamu na konci kompozice se mohou vyskytnou (například) tři instance ze třídy Vedení běžného účtu (klient si vede tři účty), dvě instance ze třídy Termínovaný vklad (má dva „termíňáky“) a jednu instanci ze třídy Půjčka.

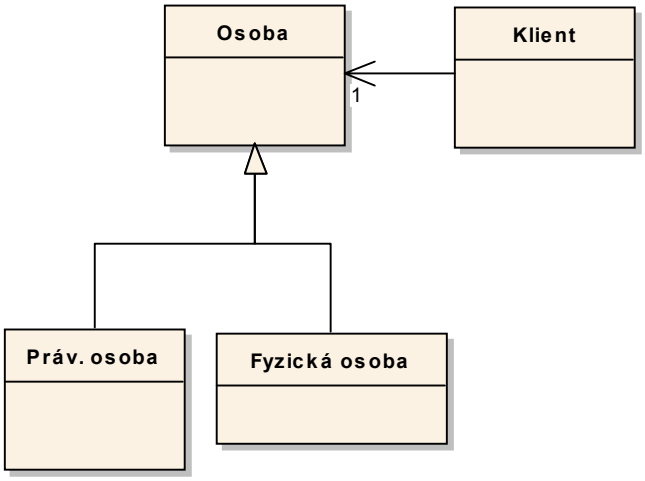
Předešlý model plně odpovídá analytickému pohledu a „zdravému selskému rozumu“. Pokud bychom hovořili s konzultantem, který neumí UML a ani jej nemusí znát, tak pro něj přeloženo do jeho řeči předešlý obrázek vyjadřuje tuto myšlenku: „Klient má N Bankovních služeb různého typu, těmito typy jsou …(následuje výčet tříd)“

Předešlá věta je zajímavá a stojí za povšimnutí. Na jedné straně se tvrdí, že Klient má N Bankovních služeb, na straně druhé se tvrdí, že jsou různého typu. Není to protimluv? Třída Bankovní služba je přece jenom jedna, tak jak můžeme tvrdit „různého typu“! Tak má Klient instance v kompozici jednoho typu (Bankovní služba) nebo vícero typů?

Odpověď zní: Oboje je správně, záleží na pohledu zvně nebo zevnitř. Zástupnost rolí totiž umožňuje, aby na konci asociace (u hvězdičky na předešlém obrázku) stály instance libovolného podtypu. Ze strany „co drží klient“ jsou role reprezentovány třídou Bankovní služba, ale v nich (v instancích) jsou dosazovány instance různých podtypů.

Uvedenou analytickou větu je třeba chápat i jako určitý vzor vyjadřování zástupnosti rolí v kompozici: „<*něco*> má N daného <*nadtypu*>, které může být různých <*podtypů*>“. Do této věty je třeba pouze dosadit výraz za příslušné <*něco*> a dosadit odpovídající výrazy také za <*nadtyp*> a za <*podtypy*>.

Podobný model postavený na téže myšlence zástupnosti rolí:



obrázek 107 Zástupnost rolí v číselníkové vazbě

Ve směru od klienta je viděna instance v číselníkové vazbě jako Osoba. V této roli může vystupovat jak Právnická osoba, tak Fyzická osoba. Tato konstrukce znamená, že osoba může být jak právnickou osobou, tak fyzickou osobou. Jinak řečeno, všude, kde do jakékoliv asociace vstupuje na konci třída Osoba, může tam hrát roli jak Fyzická osoba, tak Právnická osoba, což v tomto případě používá Klient.

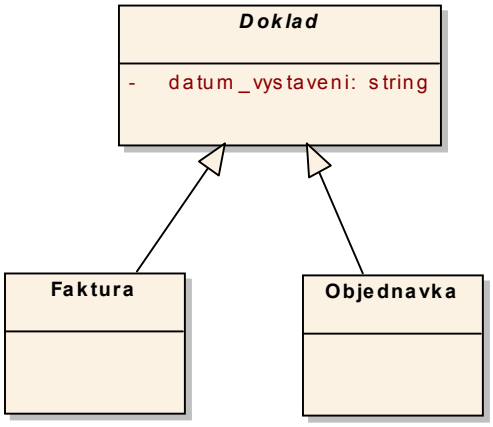
4.44 Abstraktní třída

V předešlých kapitolách jsme si uvedli, že základním posláním třídy je být kopytem pro instance, tj. třída je definicí pro vlastnosti instancí z ní vzniklých. Avšak díky existenci vztahu GEN-SPEC může nastat zvláštní situace: Některé třídy slouží pouze jako třídy v tomto vztahu na straně GEN, tj. jako předloha pro dědičnost a instance z ní vzniklé nemají smysl.

Takovéto třídy se nazývají abstraktní a jejich označení jako abstraktní vede v modelu UML již ve fázi AM k označení kurzívou. Opakem abstraktní třídy jsou třídy, které se nazývají konkrétní, z nich lze tvořit instance evidovaných informací.

Pro první příklady nemusíme chodit příliš daleko: V předešlých dvou příkladech jsou třídy Bankovní služba a Osoba typickými představiteli abstraktní třídy. Pokud klient drží N instancí bankovních služeb a začali bychom zjišťovat, jakého jsou typu, nikdy bychom nenašli v tomto seznamu nalezený typ „Bankovní služba“.

Jiný příklad na abstraktní třídu:



obrázek 108 Příklad na abstraktní třídu

Dá se říci, že abstraktní třídy jsou na tak vysoké úrovni generalizace (obecné mezi třídami), že „toho obsahují málo konkrétního“ a proto z nich nelze tvořit instance.

Prohlédněme si ještě jednou obrázek 102: Jak vypadá instance vytvořená po GEN-SPEC. Pokud je třída A („horní třída“) abstraktní, tak jediná možnost, jak se může tato třída podílet na strukturách v instancích, je jako součást instance „spodní plus horní třída“ a nijak jinak. Jinak řečeno, abstraktní třídy jsou ty třídy, které se vyskytují pouze a jenom jako „přídavek shora“ spolu s nižšími konkrétními třídami a nikdy netvoří instance samy o sobě. Nikdy se netvoří instance ze třídy A a pouze se z ní dědí.

Poznámka: Například v VB 7.0 má abstraktní třída zajímavé označení „must be inhereted“, tj. „musí být poděděna“.

4.45 Anti-vzor GEN SPEC s explozí subtříd

Vztah GEN-SPEC je čistě druhový a doporučuje se, aby byl tzv. druhově disjunktní. Toto pravidlo si nyní vysvětlíme.

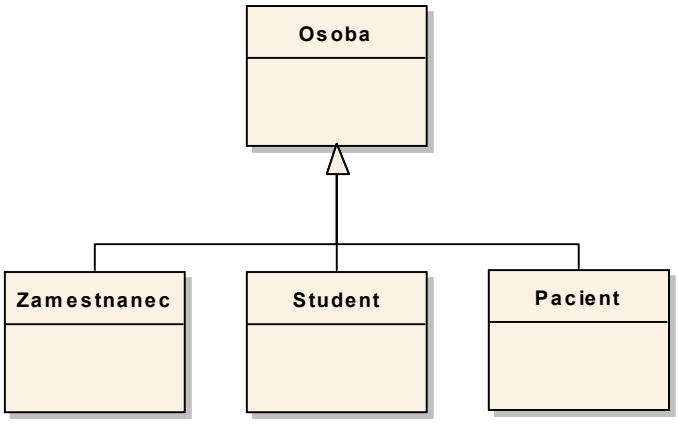
Daná instance spadá buď do jednoho druhu nebo do jiného druhu. Nedodržení této zásady vede k tzv. explozi subtříd (*„explosion of subclassing“)*, tj. k prudkému rozmnožení počtu tříd vzniklých kombinatorickým křížením druhů. Někdy toto chybné řešení vede i k nepříjemnému problému putování instance ze třídy do třídy, kdy se vyžaduje aby instance do určité doby byla z jedné třídy a od určité doby z jiné třídy.

„Anti-vzor exploze subtříd“, tj. „odstrašující vzor“, který by se neměl používat, si uvedeme následujícím příkladem:

Budeme navrhovat informační systém pro fakultní nemocnici vysoké školy.
V evidenci se budou vyskytovat osoby. Zavedeme proto třídu Osoba, která nese jméno, příjmení a rodné číslo. V systému budou vystupovat „speciální“ osoby, jako jsou:

- Zaměstnanec, na něj se váží údaje jako plat, karta zaměstnance apod.
- Student, na něj se váží údaje jako prospěch, ročník apod.
- Pacient, na něj se váže chorobopis, lůžko - kde leží, léčebné procedury apod.

Zdá se být logické, že třídu Zaměstnanec chápeme jako specializaci třídy Osoba (zaměstnanec je vlastně osobou), podobně také Student a Pacient (jsou také osobou). Takže zavedeme strom GEN-SPEC např. takto:

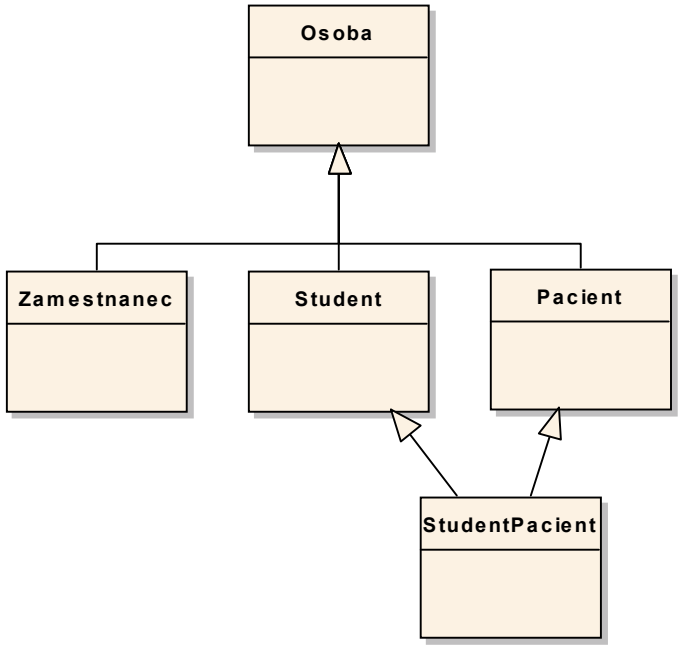


obrázek 109 "Dědicové" osoby ve vztahu GEN-SPEC

Vše vypadá vcelku rozumně, protože jak Zaměstnanec, tak Student, tak Pacient „umějí“ to, co Osoba, protože vlastnosti „podědily“ (viz obrázek 102).

Vše platí do té doby, než budeme chtít v evidenci podchytit situaci, kdy Student je například na horách, zlomí si nohu a jde si lehnout do „své fakultní“ nemocnice na lůžko jako Pacient. Co s tím musíme udělat v evidenci? Najednou daná evidovaná instance musí mít vlastnosti jak ze třídy Student, tak ze třídy Pacient.

Syntaxe UML dovoluje násobný GEN-SPEC, tedy nabízí se určité řešení, ale sami cítíme, že to „není ono“:



obrázek 110 Začátek křížení subtříd a počátek jejich exploze

V tomto případě třída StudentPacient „umí“ být jak Pacientem, tak Studentem, protože „podědí“ vlastnosti od obou tříd. Nastává tzv. násobný GEN-SPEC.

Touto konstrukcí ale vznikají nečekané problémy:

- musíme daného studenta „přestěhovat“ z „typu do jiného typu“, v tomto případě ze Studenta do StudentPacient
- dochází k explozi subtříd křížením, jak si ukážeme
- samotné křížení subtříd vede k nejednoznačnostem, koho s kým křížit, jak si ukážeme

První bod snad není třeba vysvětlovat. Doposud jsme měli pouze Studenta, nyní máme Studenta Pacienta.

Druhý bod, tj. exploze subtříd si vysvětlíme takto: Nechť někteří Studenti současně v nemocnici pracují jako zaměstnanci a chceme to zaevidovat. V evidenci tak vzniká třída kříženec ZaměstnanecStudent. Navíc někteří Zaměstnanci mohou být Pacienty (uklízečka byla taky na horách a zlomila si nohu), vzniká tak třída kříženec ZaměstnanecPacient. Dostáváme tak všechny možné dvoj-kombinace dědiců tříd podle potřeb všech kombinací.

Ale tím problém nekončí: Co když potřebujeme zavést v evidenci Studenta, který si zlomil nohu, stává se Pacientem, a současně pracuje v nemocnici jako Zaměstnanec? Vzniká zajímavá otázka: Co je správně, prokřížit tři vrchnější třídy Student - Zaměstnanec - Pacient anebo použít některou z dvojic tříd a k nim přidat třetí třídu? To není tak jednoznačné, která kombinace má přednost! A to je myšleno oním třetím bodem „nejednoznačnost prokřížení".

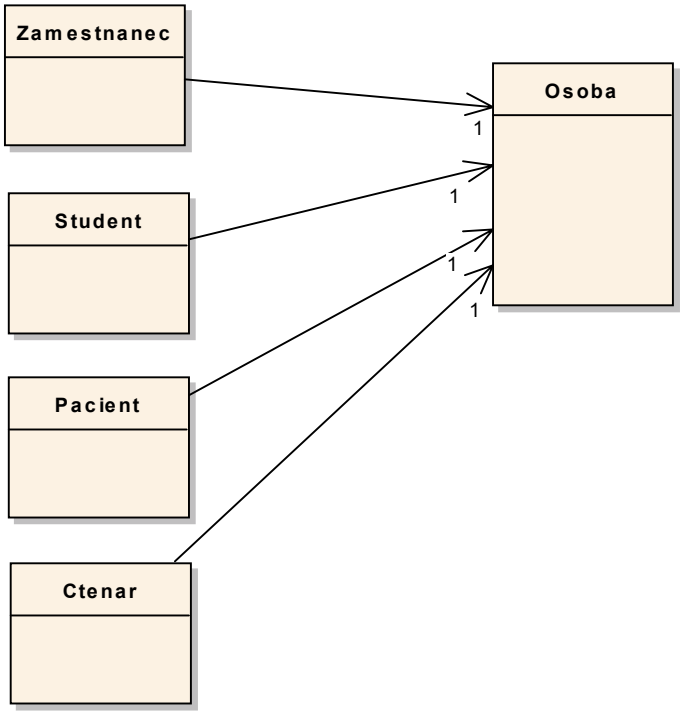
Navíc ještě netušíme, co nás jako analytiky čeká na další schůzce s budoucím uživatelem systému, s naším konzultantem. Dotyčný nám sdělí: „Minule jsme o tom nehovořili, ale máme ve fakultní nemocnici také knihovnu a budeme chtít evidovat čtenáře, což jsou taky osoby. K nim, tj. ke čtenářům, budeme evidovat půjčené knížky, upomínky a tak podobně." Jako analytici vidíme, že tak vznikne nová třída Čtenář. S hrůzou v hlase se zeptáme: „ A kdo všechno může být Čtenářem?" Odpověď zní: „Kdokoliv, Pacient, Student, Zaměstnanec, anebo jen tak Osoba…" Kombinatorika je v tomto případě příliš neúprosná a museli bychom prokřížit hodně divoké kombinace!

Ten model je prostě špatně. Jak tedy vypadá ten správný model?

V tomto případě byla narušena disjunkce typu. Vztah GEN-SPEC má totiž jednu základní vlastnost, na kterou nesmíme zapomínat: Dvě instance, které vzniknou ze dvou vedle sebe stojících potomků se společným předkem (např. instance ze třídy Student a druhá instance ze třídy Zaměstnanec) o sobě nic neví. Instance ze dvou tříd potomků nejsou díky GEN-SPEC v interakci, protože v GEN-SPEC interagují třídy jako typy, tj. „poskládají se typy a ne instance".

My však potřebujeme něco jiného než skládat typy! Potřebujeme, aby pokud držíme instanci Studenta a současně je tento Student evidován i jako Zaměstnanec, aby jak tato instance ze třídy Student, tak instance ze třídy Zaměstnanec měly stejnou hodnotu rodného čísla. Jakmile se hovoří o stejné hodnotě rodného čísla jako o hodnotě, musíme jako analytici zbystřit pozornost! Jaká je to úroveň z hlediska meta, třídy? Mít stejnou hodnotu znamená sdílení hodnot a tedy instancí a nikoliv tříd! Musíme si ukázat na tutéž instanci Osoby s daným stejným rodným číslem. Tuto vlastnost však GEN-SPEC díky disjunkci typů nezná, znamená to, že GEN-SPEC není pro tento model vhodná interakce.

Závěr: Potřebujeme nasdílet instanci Osoby s daným rodným číslem a nikoliv podědit vlastnosti tříd. Musíme tedy nahradit interakci GEN-SPEC takovou interakcí, která vede ke sdílení instancí a tou je, jak víme, běžná asociace. Znamená to, že Zaměstnanec (a podobně další třídy původně dědicové) umí být osobou nikoliv proto, že z ní dědí (nevhodná interakce!) ale proto, že si ukazují na instanci ze třídy Osoba. Jedná se takto o interakci mezi instancemi zavedenou takto:

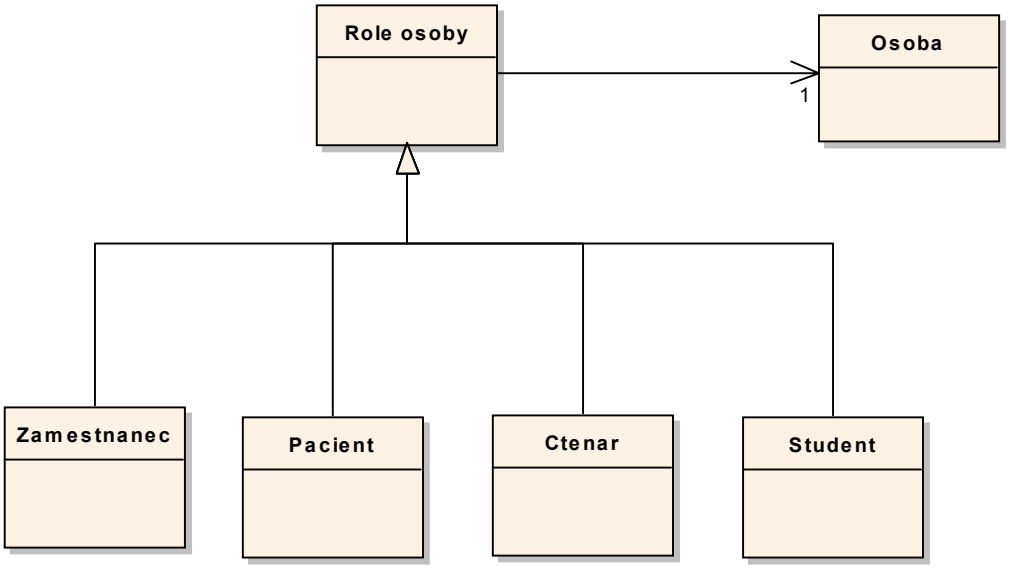


obrázek 111 GEN SPEC je nahrazen běžnou asociací

Jak v tomto modelu evidujeme instance ze tříd Student, Zaměstnanec, Pacient a Čtenář? Každá z těchto entit má svůj seznam instancí a každá z těchto instancí z těchto tříd si ukazuje na nějakou instanci ze třídy Osoba.

Je zřejmé, že nyní nenastává kolize: Pokud evidujeme někoho jako Studenta a současně jako Zaměstnance, tak bude existovat taková instance v seznamu Student, která si bude ukazovat na tutéž instanci Osoby, na jakou si ukazuje nějaká instance ze třídy Zaměstnanec. Takto může samozřejmě vzniknout kdykoliv jakákoliv „divoká kombinace“. Například pokud máme v evidenci někoho, kdo je „vším“, tj. čtenářem, pacientem, studentem i zaměstnancem současně, budou ve všech seznamech existovat instance, z nichž každá si bude ukazovat na tutéž instanci ze třídy Osoba.

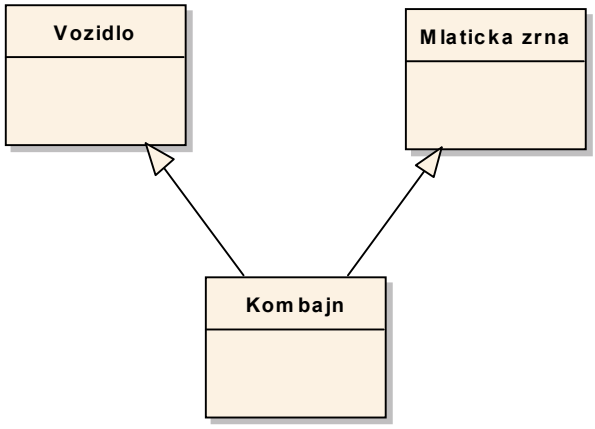
Tento model ještě nemusíme považovat za konečný. Při pohledu na předešlý obrázek zjistíme, že všechny třídy jak Zaměstnanec, Čtenář, atd. mají „něco společného“ a tím je ukazování si na instanci ze třídy Osoba. Tuto vlastnost můžeme „vytknout“ nahoru do vztahu GEN SPEC a vznikne tak jedna třída nad všemi těmito třídami. U následujícího obrázku se nedejme zmýlit podobností s první chybným návrhem, jedná se o diametrálně odlišnou situaci: „Nahoře“ není Osoba, ale jiná třída a z ní teprve vede „ukazatel“ na instanci ze třídy Osoba :



obrázek 112 Lepší model bez exploze subtříd

Další úvaha může vést ještě k otázce, zda je daná běžná asociace z Role Osoby do Osoby pouhou číselníkovou vazbou anebo vztahem k parentovi. Ve druhém případě bychom připustili, že Osoba „zná“ typ Role osoby (o této problematice viz kapitola *„Běžná asociace jako číselníková vazba anebo jako zpětná vazba na parenta?*“)

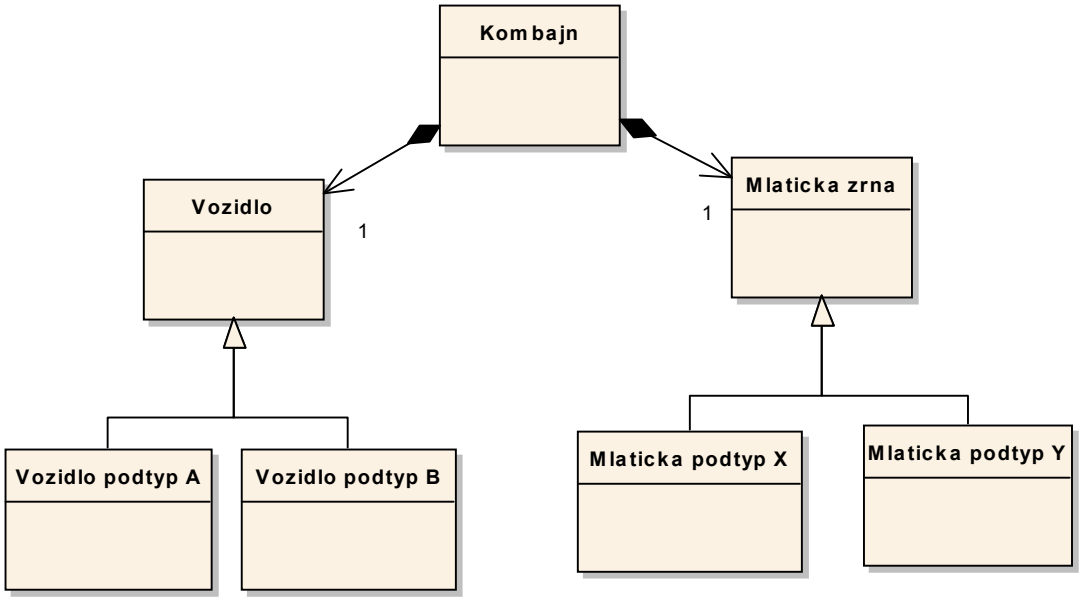
Ukažme si ještě druhý příklad, který pojednává na stejné téma, tj. jako příklad na „anti-vzor exploze subtříd“. Je znám tento příklad na násobnou dědičnost: Představme si, že máme pojem z reality jako Vozidlo a druhý pojem jako Mlátička zrna. Pojem Kombajn dostaneme tak, že z obou tříd podědíme. Jedná se logicky o „vozidlo plus mlátička zrna současně“:



obrázek 113 Kombajn chápaný jako dědic ze dvou tříd

Tato konstrukce se jeví jako logická a bude fungovat, ale pouze do určité míry. V tomto příkladu může být totiž skryta exploze subtříd. Představme si, že pojem Vozidlo má nějaké své dědice, například (vymýšlíme si!) „Vozidlo podtyp A“ a „Vozidlo podtyp B“. Mlátička nechť má také nějaké dědice, například „Mlátička podtyp X“ a „Mlátička podtyp Y“. Jak teď získat Kombajn, který je současně vozidlem podtypu A a mlátičkou podtypu X? Pokud začneme křížit subtřídy, máme problém, který již známe - exploze subtříd.

Řešení je opět ve změně vazby, tj. v náhradě GEN-SPEC jinou vazbou. Zkusme navrhnout Kombajn tak, že nebude dědicem, ale bude obsahovat dvě odpovídající instance (tj. „skládá se z…“). Jedna instance v kompozici je ze třídy Vozidlo, druhá je ze třídy ze třídy Mlátička. Jinak řečeno Kombajn je „složeninou“ Vozidla a Mlátičky a proto umí oboje. Model potom vypadá nějak takto:



obrázek 114 Kombajn jako složenina nemá problémy s explozí subtříd

Díky zástupnosti rolí s podtypy daných tříd již nemáme žádné problémy s kombinacemi, protože do role předka může vstoupit kterýkoliv dědic, což samo o sobě vede k požadovaným „divokým“ kombinacím.

Oba příklady popisují tentýž problém a také oba popisují řešení. Jak se tedy obecně této chybě vyvarovat?

Problém je v tom, že úvodní chybné modely vypadají na první pohled velmi dobře a bezchybně. Teprve následně se projeví problém s explozí subtříd. Nejlepší praktický postup, jak se vyvarovat této chybě, je z toho důvodu následující:

- V prvním kroku se zjistí, že úvahy o modelu vedou k dědičnosti (GEN-SPEC).
- Ihned se tento model vyzkouší z pohledu možných kombinací dědiců a položí se otázka: Vedou tyto kombinace k explozi subtříd? Na tuto otázku je třeba odpovědět co nejdříve.
- Pokud se v předešlém bodě na otázku odpoví ano, vztah GEN-SPEC se změní na vztah ASSOCIATION (o vztazích viz *obrázek 99*), tj. na vztah mezi instancemi. Musí se rozhodnout, zda se jedná o vztah kompozice resp. agregace (složenina) obdobná „problému kombajn“ anebo zda se jedná o běžnou asociaci (obdoba problému „osoby ve fakultní nemocnici“).
- Záměnou vztahu GEN-SPEC vztahem ASSOCIATION problém odpadne.

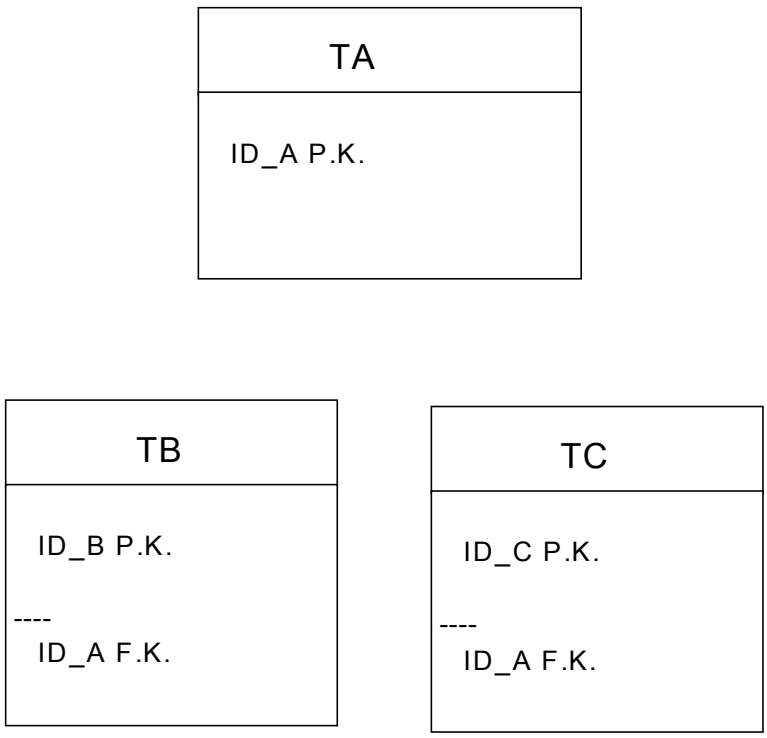
Poznámka: V řešení problému exploze subtříd je mimo jiné ukryt vzor BRIDGE, viz přiložená kniha „DESIGN PATTERNS v OOP“ .

4.46 Mapování GEN-SPEC do relační databáze podle vzoru „čisté mapování 1:1“

V aplikační úrovni hybridních systémů se při přechodu do designu mapuje GEN-SPEC na dědičnost (inheritanci) v daném OOP jazyce. Otázkou je, jak se GEN SPEC mapuje do odpovídajících struktur relační databáze.

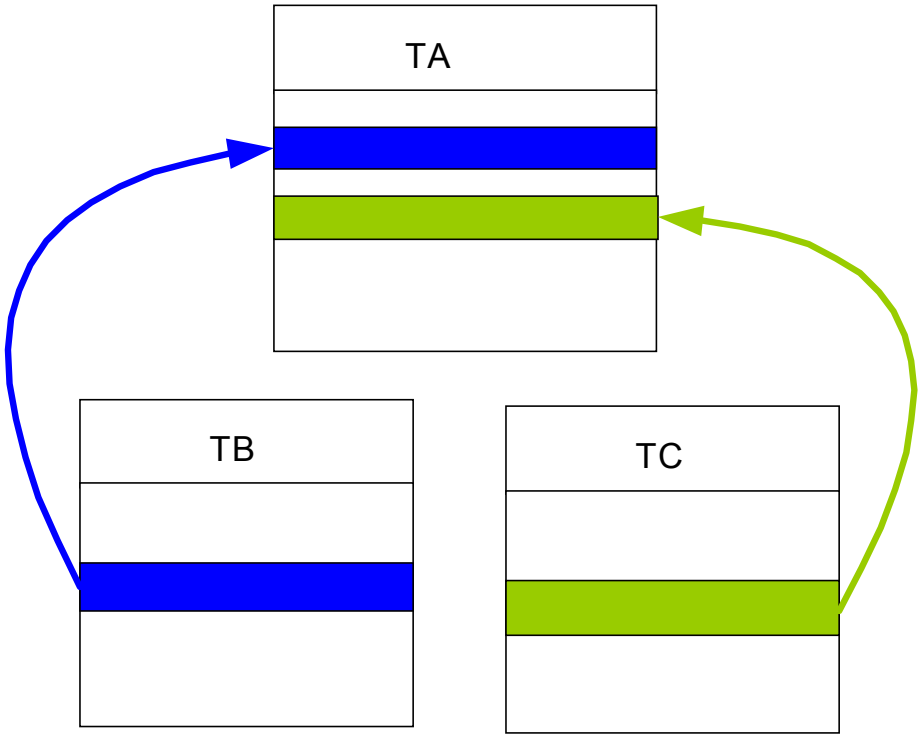
Existuje několik možných způsobů mapování GEN-SPEC do RDB, z nichž jeden je výjimečný tím, že je „čistý“, tj. bez optimalizačních kroků. Všechny ostatní postupy mapování GEN-SPEC do relační databáze se vyznačují tím, že se designér rozhodne „rozpustit“ nějakou entitu a ubude tak počet tabulek. Při těchto postupech počet tabulek neodpovídá počtu analytických tříd. Při „čistém“ mapování se počet analytických entit rovná počtu tabulek, protože každá entita má svou tabulku.

Nechť tedy analytik odevzdal model podle obrázku obrázek 101. Designér se rozhodl, že tento analytický model bude mapovat do struktur relační databáze „čistým mapováním 1:1“. Znamená to, že každá třída bude mít svůj obraz v RDB, vzniknou tedy tři tabulky, označme je TA (předek), TB (jeden dědic), TC (druhý dědic). Každá tabulka má svůj ID primární klíč (P.K.), nazvem je podle schematu ID_<název třídy>, tj. po řadě ID_A, ID_B a ID_C. Otázkou je, jak bude vypadat putování cizího klíče. V tomto případě putuje klíč z tabulky TA do obou tabulek TB a TC takto:



obrázek 115 Struktura tabulek vzniklých z GEN SPEC "čistým" mapováním

V tomto případě je vhodné popsat, jak jsou přes cizí klíč provázány záznamy:



obrázek 116 Provázání záznamu v RDB vzniklých z GEN SPEC

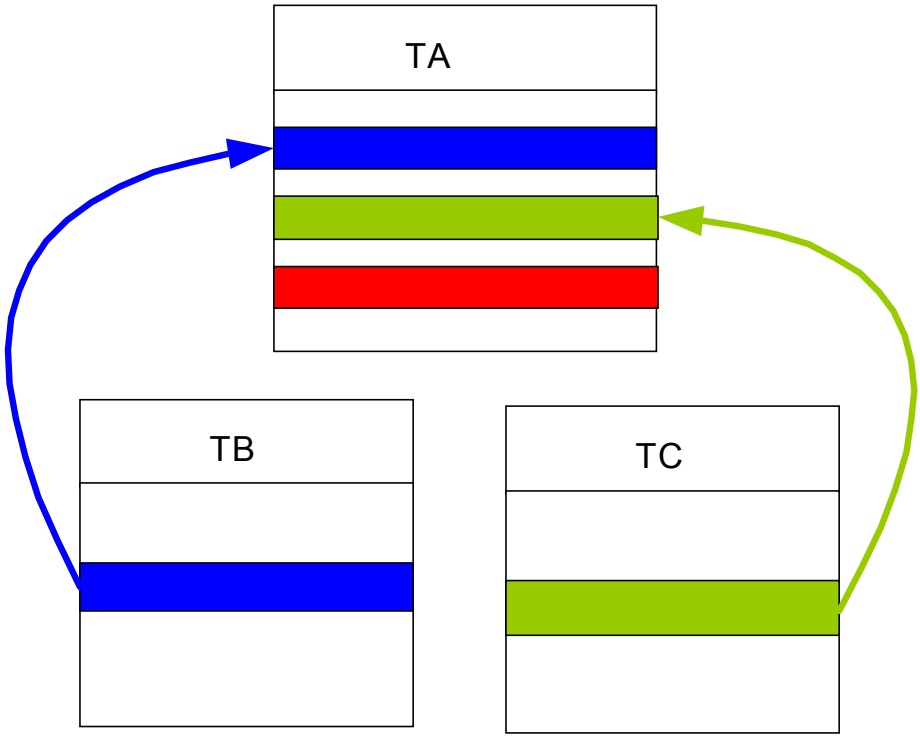
Tento obrázek je velmi podobný jako obrázek 103. Rozdíl je v tom, že obrázek 116 popisuje konkrétní strukturu, která vznikla jako obraz perzistence instancí v RDB (instance v konkrétním designu relační databáze).

Předešlý obrázek je třeba číst takto: Analytická instance ze třídy B má svou perzistenci realizovanou v RDB (to už je design) tak, že část instance je uložena v tabulce B a část v tabulce A. „Celý záznam“ instance ze třídy B se tedy čte „odspodu“, nejprve v tabulce B plus to, co je v tabulce A (viz modrá barva na obrázku). Propojení se děje přes cizí klíč ID_A. Podobně totéž platí pro instanci ze třídy C (viz zelená barva na obrázku).

Všimněme si, že u obou instancí „modré“ tak „zelené“ je vždy jejich „horní“ část uložena ve stejné tabulce A. To je vlastně obrazem GEN-SPEC v tom smyslu, že tato část instancí se ukládá do téhož formátu dat (společná obecná část).

Zajímavá otázka: Jak se na předešlém obrázku projeví to, že třída A je abstraktní anebo že není abstraktní, tj. že je konkrétní?

Pokud je třída A konkrétní, tak bude v tabulce TA existovat záznam, který se „nečte odspodu“ z tabulek TB nebo TC, ale přímo od úrovně tabulky TA takto:



obrázek 117 Záznam, který se "nečte odspodu" (červeně)

U „červeného“ záznamu ať hledáme, jak hledáme, tak v žádné „spodní“ tabulce nenalezneme takový záznam, který by dával dohromady celý „červený“ záznam odspodu nahoru podobně jako záznam „zelený“ nebo „modrý“. „Červený záznam“ je obrazem toho, že žije analytická instance ze třídy A, pracuje se s ní a tato instance potřebuje perzistenci. Její perzistenci vidíme jako „červený záznam“ pouze v tabulce A.

4.47 Zásada „použití ve směru odspodu nahoru“ a zásada „odstínit vrch“ ve vztahu GEN-SPEC

Při mapování z analytického modelování do designu by se měly dodržet ty vlastnosti daného vztahu, které jsou pro daný vztah charakteristické. V případě GEN-SPEC je důležité dodržet tyto dvě zásady

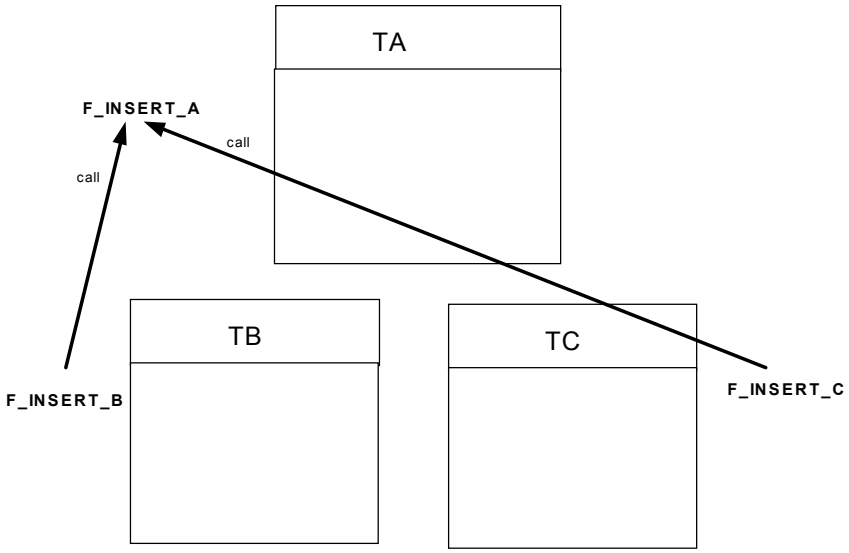
- zásada použití ve směru odspodu nahoru (každá spodní třída používá horní třídu a nikdy naopak)
- zásada odstínění implementace vrchu, což znamená, že prvky definované v horní třídě lze použít pouze přes definovaný vnější obal v horní třídě a ne přímou viditelností uvnitř implementací tříd. Na obrázku viz obrázek 103 tomu odpovídá červená dvojšipka zakazující viditelnost dvou částí vnitřku instance, kdy jediná možnost v použití zespodu nahoru spočívá pouze ve volání obalu jako zelená šipka.

První zásada vede ke kýženému efektu opětovné použitelnosti, kdy dědicové používají vrchního předka. V designu programu hybridního systému to znamená, že z pohledu databázisty se funkcionality nad tabulkou TA programují pouze jednou. Tyto funkcionality totiž odpovídají třídě A v analytickém modelu, která je v GEN-SPEC „zespodu používána a svrchu nepoužívá“. Navíc díky dodržení směru striktně „odspodu nahoru“ se přidání nebo změna u nějakého dědice nedotkne žádného z již existujících dědiců a ani vrchní třídy.

Druhá zásada nazvaná „odstínění implementace vrchu“ zvyšuje flexibilitu a stabilitu programu. Pokud naopak a chybně není implementace horní třídy A odstíněna od spodní třídy (tj. spodní třída používá implementaci definovanou v horní), tak každá změna uvnitř horní třídy (například vývojem, změnovým řízením apod.) se ihned projeví ve všech dědicích, kteří musí být také „překopáni“. Pokud je horní třída odstíněna (a program je navržen dostatečně čistě), tak velmi mnoho změn v horní třídě se ve spodních třídách vůbec projeví.

Například nechť existují tři funkcionality: F_INSERT_A, F_INSERT_B a F_INSERT_C, což mohou být funkce anebo metody objektů. V těchto funkcionalitách jsou uschovány na nejnižší úrovni designu příkazy INSERT do odpovídajících tabulek. Zvolíme konstrukci, že „spodek volá vrch a neví, co se uvnitř děje (nevidí

implementaci)“. V našem případě to znamená, že ukládání instance ze třídy B probíhá tak, že funkce F_INSERT_B (resp. metoda objektu) nejprve zavolá funkci F_INSERT_A a přitom neví, co se v ní děje. Teprve potom se provede uložení do B. Totéž pro C, znázorněno podle obrázku:



obrázek 118 Volání funkcionality zespodu nahoru s odstíněním implementace

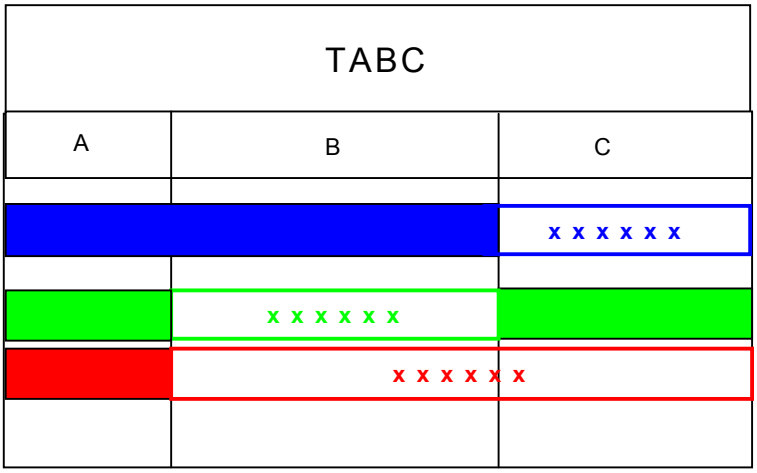
Při tomto „čistém“ řešení pochopitelně pokud někdo například přidá nebo ubere sloupec v tabulce TA, tak sama funkce F_INSERT_B (nebo F_INSERT_C) tuto změnu nepocítí. Ale to je ještě jednoduchý příklad: Uvědomme si, že s celou třídou A mohou spojeny další a další vztahy, do kterých třída vstupuje, může být dědicem jiné třídy, může obsahovat další svoje složeniny v kompozici atd. To vše a změny s tím spojené by při odkryté implementaci znamenalo neustále překopávat potomky.

4.48 Mapování GEN-SPEC do RDB podle vzoru „kočkopes“

Velmi častý způsob mapování modelu tříd analytického modelování do designu relační databáze je následující:

Nechť analytik odevzdal model podle obrázku viz obrázek 101. Designér již vyčerpal všechny své znalosti a použil všechny možnosti (indexy apod.) a přesto vazba mezi tabulkou „potomka“ a „předka“ vede stále k technologickým problémům. Rozhodne se proto zbavit se této vazby a provede nejčastější optimalizaci a vytvoří „kočkopsa“. Z původních tří entit A, B, C vytvoří jednu tabulku, označme ji jako TABC. V ní budou

skupiny sloupců, které odpovídají původním entitám, tyto sloupce označme odpovídajícími skupinami A, B, C podle tříd, ze kterých vznikly. Dostáváme tak následující obrázek, který zobrazuje, jak vypadá struktura tabulky TABC a také jak vypadají odpovídající záznamy vzniklé ze tříd A, B, C podobně jako v předešlé kapitole:



obrázek 119 Mapování vztahu GEN-SPEC do RDB podle vzoru "kočkopes"

Záznam vzniklý z instance ze třídy B (modrý) používá sloupce A a B, přičemž sloupce C jej nezajímají, záznam vzniklý ze třídy C (zelený) používá sloupce A plus C, sloupce B jej nezajímají. Pokud je třída A konkrétní, mohou existovat záznamy, které vznikly z instance ze třídy A (červené), používají sloupce A, ostatní sloupce B i C jej nezajímají.

Tento vzor nazvěme vzor „kočkopes“, protože tento název vystihuje jeho podstatu. Pokud zvolíme za vrchní třídu Zvíře a jeho dva dědice Kočka a Pes, tak tabulka, která vznikne tímto postupem, je opravdový a nefalšovaný „kočkopes“.

Co se týče výhod tohoto postupu mapování, tak existuje jediná výhoda a je dána důvodem, proč je vlastně tento postup zvolen. Tímto důvodem je odstranit pomalost zpracování vazby mezi tabulkami. Porovnejme tento obrázek s obrázkem „čistého“ mapování např. obrázek 117, kde odpovídají také barvy záznamů z daných tříd. Vidíme, že v případě „kočkopsa“ načtením jednoho řádku získáme ihned údaje jak vrchní, tak spodní třídy.

Nevýhod postupu podle vzoru „kočkopes“ je trochu více. Není bez zajímavosti, že programátoři při diskusi nad tímto vzorem většinou vidí jako první nevýhody pouze čistě technické problémy, například zbytečně velká délka záznamu apod. To jsou problémy podružné. Největší nevýhodou tohoto postupu je to, že jako každá

„nečistota“ narušuje „čistotu“, tak zde je též něco „čistého“ narušeno a tím je princip opětovné použitelnosti a OOAP. To má velmi nepříjemné nežádoucí důsledky.

Jako první nepříjemný nově vzniklý problém (který samozřejmě „čisté“ mapování nezná) je to, že jak prvky ze třídy B, tak prvky ze třídy C se musejí postarat o sloupce A vždy znovu a znovu a každý prvek znovu sám bez opětovné použitelnosti. Tento problém se dá při trochu zvýšené námaze zvládnout, i když znovu opisovat části operace INSERT nebo UPDATE nad stejnými sloupci (sloupce A) není zrovna příjemné.

Horší je, pokud dojde ke změnám v entitě A. Tato změna se okamžitě týká všech prvků ze všech dědiců. Ve všech opakujících se prvcích musíme tyto sady příkazů nad databází opravit. Vrch nejenom že není vytvořen pouze jednou, ale také není dobře odstíněn.

Druhý nepříjemný důsledek souvisí s uměním typu. Jak víme, jedním ze základním principů OOAP je typovost, tj. existence třídy a instance. Díky tomu, že vytvoříme v databázi „kočkopsa“, tak v této technologii RDB, kde jsou typy reprezentovány tabulkami, jsme vlastně sloučením tabulek sloučili typy. Už nejsou tabulky TA, TB a TC, ale jedna tabulka, nazvali jsme ji TABC. Databázové prostředí pracuje se záznamem tohoto typu jako celek a nějaké vnitřní rozčlenění na skupiny sloupců je pouze pomyslné a nikoliv striktně typové. Proto má sloučení tabulek velmi nepříjemné důsledky. Otázka zní jednoduše: Co se stane, když dáme do jednoho pytle kočku a psa? Pobijí se. I zde sloučením typů vzniká nový efekt, pobití entit.

Představme si tuto situaci: Máte na starosti určitou část programu, která odpovídá třídě B. Bylo provedeno mapování podle vzoru „kočkopes“. Vše naprogramujete, otestujete, odevzdáte. Po dvou týdnech potkáte kolegu a ten vám sdělí: „Víš, že ti ten tvůj program padá?“ „Kde, prosím tě?“ zeptáte se. „Při jednoduchém INSERTU.“ „To není možné, to máme snad tisíckrát ověřeno.“ Jdete, zkusíte a opravdu, to co bylo ověřeno snad tisíckrát, opravdu padá. Začnete zkoumat a zjišťovat, čím to je … a najednou máte chuť někomu hodně od plic vynadat. Jiný kolega, který má na starosti třídu C, totiž v jednom ze svých sloupců (které vás vlastně nezajímají) nasadil podmínku constraint „not null“. Najednou vás hodně moc zajímá, co se děje ve sloupcích, které vás nezajímaly. Jdete za kolegou a řeknete mu: „Hele, vypni ten constraint prosím tě, mám s tím problém!“ A kolega vám řekne: „To nejde, já to používám…“ Výsledek asi tušíme: Budeme nuceni do tohoto sloupce dávat nějakou „dummy“ hodnotu, například 1, jenom proto, aby to nepadalo. Začíná zajímavý efekt „šamanských nepochopitelných tanců“, kdy musíme někam nějakou hodnotu dát a to úplně mimo náš problém, a to jenom proto, aby to nepadalo.

Je třeba si uvědomit, že tento efekt vznikl díky promíchání typů. Podmínky už nezní takto: V typu kočka se dějí tyto věci (platné pro typ kočka), v typu pes se dějí tyto věci (platné pro typ pes) a tyto dvě oblasti jsou striktně disjunktní. Sloučili jsme dva typy do jednoho a najednou platí: Pokud v tomto typu kočka, tak platí toto a pokud

v tomto typu pes, tak platí toto něco jiného. Nemusíme asi zdůrazňovat, jak vypadá údržba systému, který:

- jednak nemá analytický model, takže ani neznáme původní typy před sloučením,
- je plný „kočkopsů“, kdy se pojmově rozdílné entity slévají do jedné tabulky a bijí se mezi sebou
- navíc je plný „šamanských nepochopitelných tanců“ jako důsledek předešlého bodu: Něco se musí nutně obsloužit, i když to nepatří k danému problému řešení, případně provádět přepínání ve smyslu „pokud je to moje…“

Je třeba podotknout, že tato optimalizace „kočkopsa“ je velmi častá. Pokud se provádí se znalostí věci, existuje původní analytický model a ví se, jak vznikají „kočkopsi“, tak nenastávají v systému vážné problémy, pouze je pracnější jeho údržba.

Například když analytik odevzdá model, ve kterém se dva dědicové „liší velmi málo“ (například ve dvou atributech), tak s největší pravděpodobností designér do RDB provede úpravu podle vzoru „kočkopes“ a „slije“ obě dvě entity do jedné tabulky. Musí však vědět, co činí a s jakými důsledky. Největší problémy vznikají při tvorbě „kočkopsů“ rovnou designem bez rozumných analytických myšlenek a modelů a to přímo lepením datových struktur způsobem „padni kam padni“. V systému pak nalezneme tabulky pro „firmobčanoklienty“, „úvěro-termíňáky“ apod. Takový systém se stává silně nečitelným.

4.49 Mapování GEN-SPEC do RDB „anti-vzorem přetížení sloupců“

Další způsob mapování vztahu GEN-SPEC si uvedeme jako „anti-vzor“, tj. jako postup, který „se vřele nedoporučuje“. Viděl jsem jej použit v několika firmách a vždy měl katastrofální následky.

V prvním kroku se použije vzor mapování „kočkopes“ a sjednotí se tabulky ve vztahu GEN-SPEC. Jako pěkný příklad si můžeme vzít strom dědičnosti pro Bankovní služby, jak jej ukazuje obrázek 106. V tomto případě se designér rozhodne použít jednu tabulku TSLUZBA a umístí do ní všechny sloupce ze všech entit (Úvěr, Půjčka atd.). Přitom si všimne, že například v entitě Úvěr je jistina úvěru (kolik se půjčilo), v entitě Termínovaný vklad najdeme výši termínového vkladu (kolik je vloženo k úročení). Ve sloučené tabulce se obě dvě informace vyskytují jako dva sloupce a jsou téhož formátu (např. „MONEY“), přičemž se vždy používá vždy v daném

záznamu jen jedna z nich. Designér se proto rozhodne tyto dva sloupce nahradit jedním sloupcem a vznikne tak jeden sloupec typu peníze, který

- pokud se jedná o úvěr, má význam jistiny úvěru
- pokud se jedná o termínovaný vklad, má význam výše vkladu

Samozřejmě tento „univerzální sloupec“ se nemůže jmenovat ani jistina a ani výše vkladu, ale nějak jinak, například PARMONEY1. Takovýchto parametrických „peněz“ najdeme v bankovních službách hned několik a tak se zavede například šest polí PARMONEY1 až PARMONEY6. Dále se zjistí, že totéž platí o datumech, protože existují opravdu všechny možné datumy ve službách. Proto se zavedou pole PARDATUM1 až PARDATUM10. A tak dále. Výsledkem je, že existuje jedna tabulka s jakýmisi univerzálními sloupci, jejichž význam je dán typem služby.

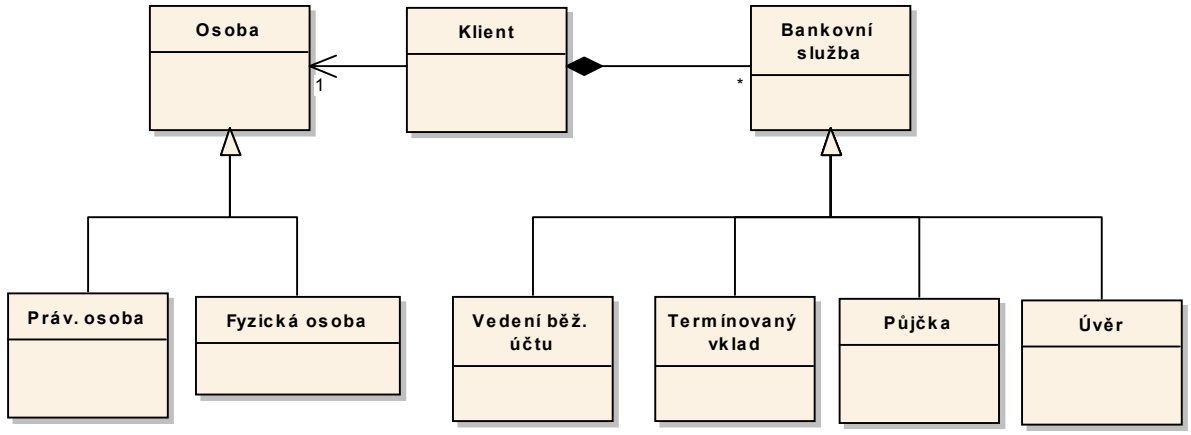
Na první pohled tato konstrukce vypadá „chytře“, protože se zdá, že nemusíme pro novou službu zakládat novou tabulku a proto se návrh jeví jako flexibilní. Bohužel toto očekávání tento model v žádném případě nesplňuje. Uvedená univerzálnost by platila tehdy, kdyby i všechny algoritmy nad službami byly také univerzálně platné pro všechny služby, což není pravda. Přece jen s informací jistina úvěru se pracuje určitě jinak, než s výší vkladu. Jediné, co tímto postupem ušetříme, je námaha s novým CREATE TABLE (přidání tabulky při čistém mapování) nebo ALTER TABLE (přidání sloupců při „kočkopsovi“). Ušetříme tak počet sloupců, ale entity se chovají podle svých pravidel, které musíme naprogramovat tak jak tak. V jednom případě budeme v programu oslovovat daný sloupec jako „jistina_uveru“ a při použití vzoru „přetížení sloupců“ jako „PARMONEY1“, a to je jediný rozdíl.

Návrh tímto způsobem však systém přináší jiné obrovské problémy s katastrofálními následky. Mohu z vlastní zkušenosti potvrdit, že ve všech případech jeho použití ve firmách se po určité době nikdo v programu nevyznal a to dokonce po určité době ani sami autoři. Číst program s parametrickými proměnnými je velmi obtížné a dokonce při krokování nemáme jistotu, zda systém dělá to, co má. Navíc ještě začnou vznikat zajímavé chyby druhého druhu na meta-úrovni. Protože se sloupce oslovují univerzálními jmény, neexistuje typová kontrola toho, co se vlastně v systému děje. V takto navrženém systému může klidně proběhnout proces splátky úvěru nad termínovaným vkladem a typově „nic nezařve“. Typy, se kterými se pracuje, mohou být určeny dynamicky v běhu programu (obsluha vybere typ… apod.). Chyba v hodnotě proměnné vede k záměně typu a mohou se spustit procesy, které danému typu vůbec nepatří. Dovedeme si představit, jakou paseku udělá spuštěný proces splátek úvěru nad termínovanými vklady! A jak to potom opravit a vrátit zpátky (pokud se operace již odsouhlasí a tzv. „komitne“)! Tento způsobe mapování osobně nemám rád, připadá mi totiž jako už příliš velký odklon od principů OOAP.

4.50 Použití vzoru „číselník typů“ ve vztahu GEN-SPEC

Pokud analytik použije vztah GEN-SPEC a v modelu se mu objeví odpovídají „strom dědičnosti GEN-SPEC“, měl by použít určitou konstrukci jako vzor, kterou si nyní vysvětlíme na příkladu a poté tento příklad zobecníme do vzoru.

Představme si, že analytik po rozhovoru s konzultantem dospěje k následujícímu modelu:



obrázek 120 Model osoba, klient a bankovní služby

Tento model plně využil zástupnosti rolí. V roli toho, co drží klient v kompozici (jeho bankovní služby) mohou být instance různého podtypu bankovních služeb (heterogenní seznam). Totéž platí i na druhé straně, kdy do role Osoby mohou typově vstoupit jak instance fyzické osoby, tak instance právnické osoby.

Tento model stojí ještě za určitou diskusi. Všimněme si, že model má dvě úrovně, horní a dolní, čteno shora dolů. Na horní úrovni, tj. když se zaměříme pouze na entity Osoba, Klient a Bankovní služba, tak vztah mezi nimi je oddělen od vztahů dvou stromů dědičnosti. Na horní úrovni se modelem jednoduše sděluje věta: Klient má N Bankovních služeb a Klient má za sebou Osobu, kterou reprezentuje. Všimněme si, že například změny ve stromech dědičnosti se této věty nikterak nedotknou. Například přidat novou Bankovní službu znamená „přidat a začlenit potomka do rodiny Bankovních služeb“.

Uveďme si další zajímavou skutečnost. Na tomto příkladu je krásně vidět, v jakém vztahu je analytické modelování a náhled uživatele: Všimněme si, že grafický zápis na předešlém obrázku plně odpovídá „laickému“ pohledu uživatele a „zdravému selskému rozumu“. Věta laika „Klient má N Bankovních služeb různého typu“ je na obrázku přímo patrná. Navíc, pohled na tento model odpovídá také přesně pozicím všech informací, o kterých uvažujeme pohledem laika a „zdravým selským rozumem“. Podívejme se na předešlý příklad a zkusme se například zeptat: Kam patří IČO právnické osoby? V dobře položené otázce je skryta i odpověď: IČO právnické osoby leží v právnické osobě (to je až primitivně hloupá věta!) a do této třídy jej umístíme. Podobně: Kam patří rodné číslo fyzické osoby? Odpověď je v otázce: Rodné číslo fyzické osoby leží ve fyzické osobě a do této třídy jej umístíme. Podobně: Kde budeme hledat jistinu úvěru, výši termínovaného vkladu, zůstatek běžného účtu atd.? Odpovědi slyšíme přímo v otázkách. Na první pohled se to jeví jako „tautologie“, ale všimněme si, že tato shoda platí i v náhledu na strom dědičnosti, tj. na vztah GEN-SPEC. Zkusme se zeptat: Kam umístíme datum založení bankovní služby? Vidíme, že pokud použijeme metodu „jak slyšíme, tak odpovíme“, dochází ke shodě s logikou věci: Datum založení bankovní služby je atributem bankovní služby, tj. leží v generalizující entitě Bankovní služba, ze které ostatní dědí. Tak to namalujeme a tak to také chápeme.

Další zajímavá věc, kterou si můžeme na tomto modelu ukázat, je pohled designéra: Pokud provede mapování jedna ku jedné, daný model lze chápat ihned jako model databáze a vidíme i putování cizích klíčů. Stačí pouze použít již zavedené vzory (putování klíče ve směru proti šipkám, v GEN SPEC směrem dolů a v kompozici ku N od majitele ke vlastněným prvkům).

Dále si můžeme uvést, jaké všechny zásahy může designér provést a „degenerovat“ analytický model do tabulek mapováním nikoliv 1:1. Může zavést „kočkopsy“ na obou stranách (sloučí se bankovní služby a sloučí se osoby). Navíc, protože vztah mezi „kočkopsem“ Osoba (fyzická i právnická osoba dohromady) a klientem je ku 1, tak může sloučit i tyto dvě tabulky a dostane tabulku firmo-občano-klienta.

Zmiňme se ještě o možnosti přetížení sloupců v bankovních službách a v osobách (rodné číslo a IČO jako jeden sloupec apod.)

To vše, co tu teď píšeme, je pouze úvodem, který shrnuje již probrané kapitoly. Cílem tohoto příkladu je ukázat novou konstrukci a zavést vzor, který je velmi důležitý a užitečný.

Pokračujme v příkladu: Konzultant se s tímto modelem seznámí v tom smyslu, že je mu vysvětlen slovně a následuje tento jeho dotaz:

„Víte, některé druhy bankovních služeb nemohou být poskytnuty některým druhům osob. Například půjčka občanovi nemůže být poskytnuta právnické osobě a úvěr pro právnickou osobu nemůže dostat fyzická osoba. Tento model, tak jak jej máme před sebou, ptám se: Provádí toto omezení ano nebo ne?“

Odpověď pochopitelně zní: „Nikoliv.“ Bráno čistě technicky zástupnost rolí funguje tak, že libovolný dědic (potomek) může vstoupit do role předka. Jenomže konzultant prohlásí: „Ale my toto omezení chceme!“

Otázka je tedy nasnadě: Jak podporovat v evidenci toto omezení?

Opět je třeba mít na paměti, že správně položená otázka dává v analytickém modelování sama o sobě odpověď. Předchozí požadavek vyslovený konzultantem zkusme přeformulovat do otázky správně položené: Konzultant vlastně chce, aby se evidovaly „povolené typy služeb k typům osob“. Tato formulace je již přesná a podle již probraného vidíme, že by tato formulace měla nějak dospět k asociativní třídě (jako povolené dvoj-kombinace podobně jako v učitel versus předmět apod.). Víme, že asociativní třída propojuje instance, ale v našem případě koho propojit s kým?

V otázce se hovoří o typech služeb a o typech osob. Máme v tomto modelu evidované typy osob a evidované typy bankovních služeb? Nedejme se zmýlit tím, že máme nějaké třídy, které odpovídají názvům těchto typů. V naší úvaze je třeba se vrátit úplně na začátek k principům analytického modelování: Evidence je vždy dána tak, že existuje nějaká třída a z ní seznam instancí. Nyní však potřebujeme seznam typů služeb a seznam typů osob. Takovýto seznam však nemáme zaveden, protože nemáme ani takové třídy. Třídy Bankovní služba a Osoba uvedené na předešlém obrázku jsou o něčem jiném: Instance z nich není seznamem typů, ale jsou to jednotlivé fyzické osoby, právnické osoby, nebo úvěry, půjčky atd. A to nejsou typy.

Nyní jsme dospěli k důležitému závěru, který řeší danou situaci: Zavedeme dva jednoduché číselníky, jeden nazvěme Typy osob, druhý nazvěme Typy bankovních služeb. Oba mají jako číselníky primitivní strukturu, např. kód, zkratka, text. Obsah těchto dvou číselníků si znázorníme takto:

Typy osob:

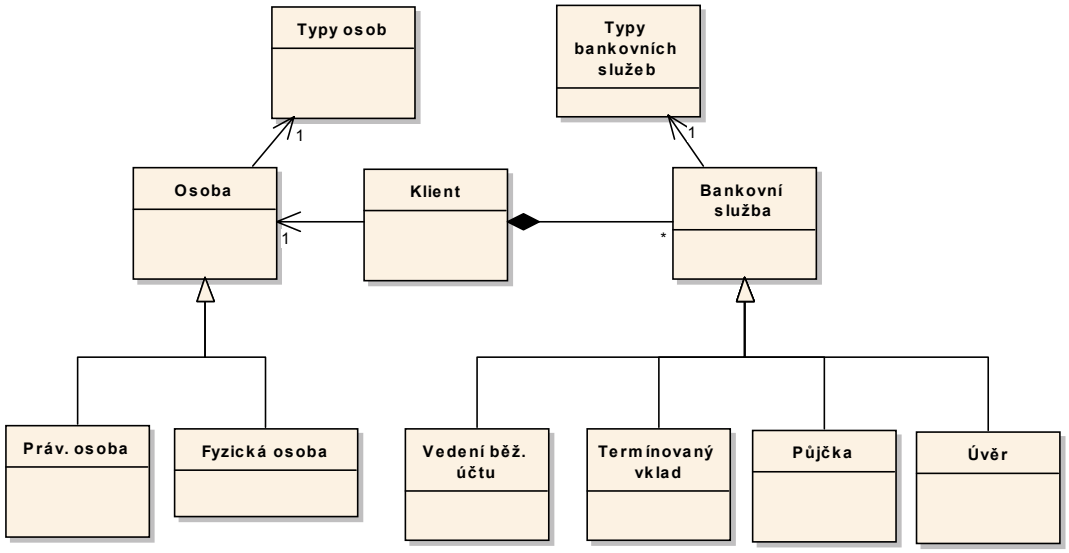
Kód	Text
1	Fyzická osoba
2	Právnická osoba

Typy bankovních služeb:

Kód	Text
1	Úvěr
2	Půjčka
3	Termínovaný vklad
4	Vedení běžného účtu

Tyto dva číselníky zavedly (opravdu primitivně) seznamy typů pro oba požadované případy. Další konstrukce je jednoduchá: Každá instance ze stromu dědičnosti osob si bude ukazovat do číselníku Typy osob na odpovídající item, a to podle typu kam patří. Znamená to, že instance ze třídy Fyzická osoba si bude ukazovat na první item číselníku Typy osob a instance ze třídy Právnická osoba si bude ukazovat na druhý item tohoto číselníku. Obdobně totéž pro bankovní služby.

Pokud si tyto dva číselníky označíme graficky, můžeme namalovat následující obrázek:



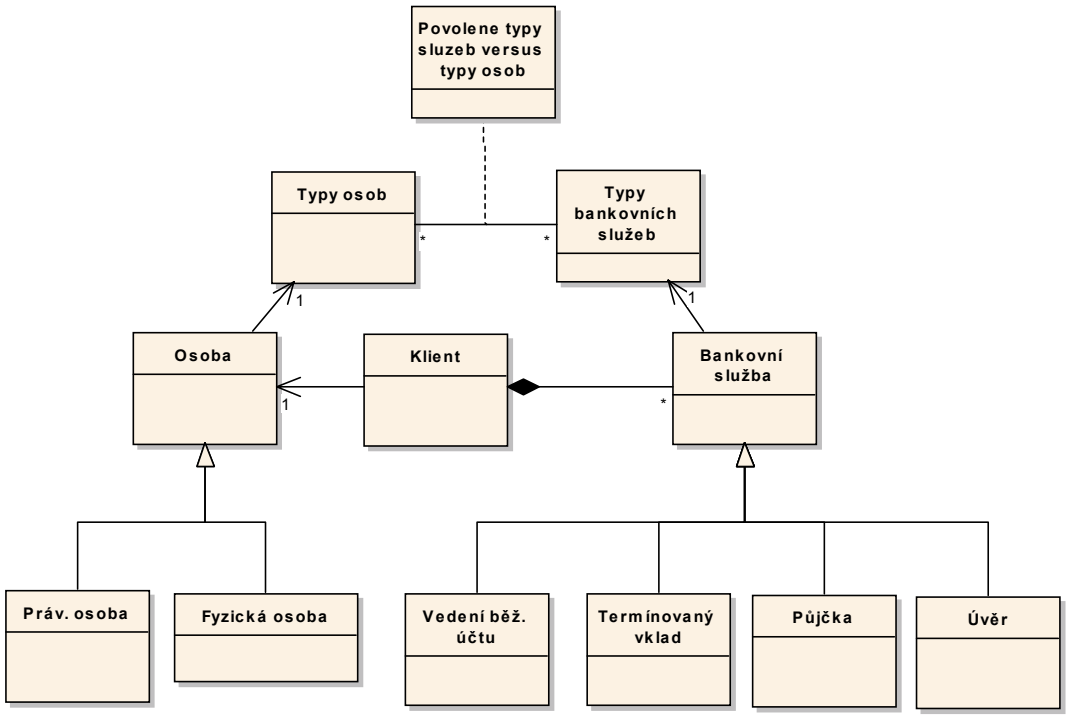
obrázek 121 Model doplněný o číselníky typů

Protože jsme číselníkovou vazbu zavedli přímo na vrcholové třídě stromu GEN-SPEC, mají tuto vazbu všechny instance ze všech odpovídajících dědiců.

Je vcelku pochopitelné, že uvedené číselníky typů nejsou „uživatelskými“ v tom smyslu, že by se do nich dalo kdykoliv „hrabat“, jak se kdy komu zachce. Mají povahu konfiguračních číselníků a jsou to vlastně „převodníky“ mezi třídou a evidovanou instancí.

Poznámka: Je třeba poznamenat, že oba číselníky na předešlém obrázku označujeme jako dvě třídy Typy osob a Typy bankovních služeb. Mohlo by se při dalším vývoji stát, že tyto dvě entity by se nám podařilo vyřešit pomocí jediné entity, protože mají stejnou strukturu a chovají se stejně. To však není pro náš vzor podstatné.

Nyní můžeme náš příklad dokončit. Je zřejmé, že hledaná asociativní třída „typy osob versus typy bankovních služeb“ propojuje zmíněné dva číselníky, což můžeme zobrazit takto:



obrázek 122 Řešení povolené typy služeb versus typy osob

Zavedení uvedených číselníků typů nám vyřešilo tento příklad, ale ukazuje se, že jsou mnohem potřebnější, než se na první pohled zdá. Jejich význam roste zejména tehdy, pokud chceme aplikaci navrhnout flexibilně. Například hodně scénářů je zahájeno slovy: „Obsluha vybere typ něčeho…(osoby, bankovní služby apod.)“. V našem modelu všechny scénáře založení nové instance do systému (tj. nová fyzická osoba, nová právnická osoba, nový úvěr atd.) by měl začít výběrem typu „co chceme založit“.

Máme dvě možnosti, jak tento výběr můžeme realizovat: Buď jej naprogramujeme natvrdo v tom smyslu, že přímo v kódu se provede výčet, například ve fromuláři pro osoby vybudujeme submenu se dvěma itemy. Obsah textů (caption) „nová fyzická osoba“ a „nová právnická osoba“ nastavíme přímo v kódu. Druhá možnost je více flexibilní: Zobrazíme obsah číselníku typů, v něm jsou zavedené uvedené texty. Obsluha vybere prvek z tohoto číselníku (nevíme, zda pracuje s prvním nebo druhým prvkem). Je třeba upozornit, že při této konstrukci se systém stane sice silně flexibilním pro přidání nebo ubrání typu, ale musíme použít některý z DESIGN PATTERNS, například PROTOTYPE nebo FACTORY.

Situace použití takového číselníku typů je natolik běžná, že doporučení vzoru zní velmi jednoduše: Je třeba vždy tento číselník typů zavést (postupem jako u typů služeb nebo typů bankovních služeb). Musíme pouze zobecnit pravidlo, jaké itemy číselníku musí číselníku typ obsahovat. V našem příkladu u osob jsou dva, u bankovních služeb jsou čtyři. Musíme tento poznatek zobecnit. Prvků číselníku bude tolik, kolik je konkrétních tříd a pro každou konkrétní třídu bude jeden prvek.

KONEC KAPITOLY ANALYTICKÝ MODEL TŘÍD

KONEC 1. dílu publikace